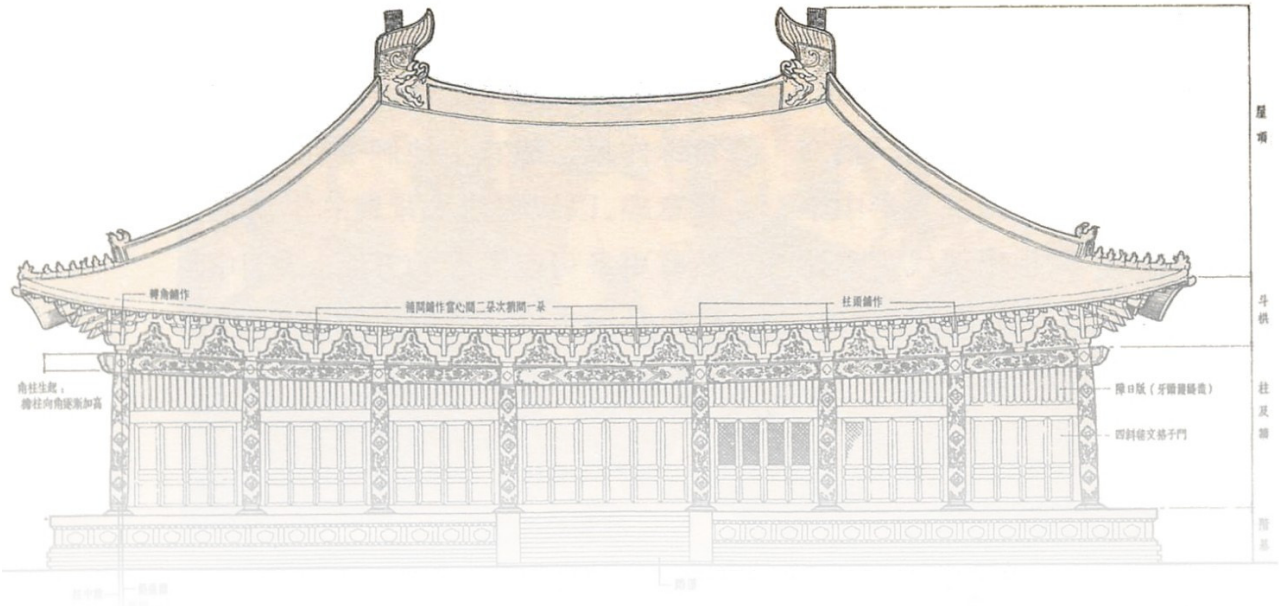
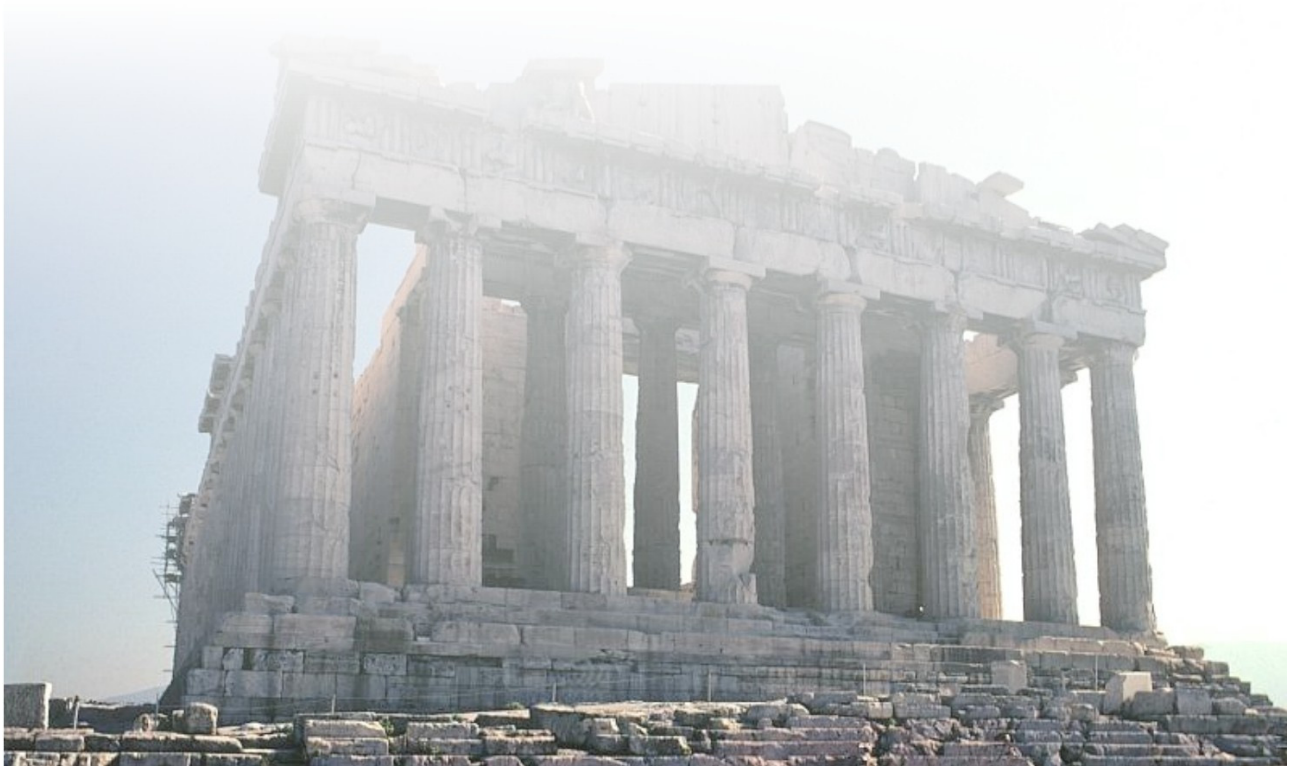


Thinking in Patterns with Delphi

Delphi 模式编程 电子版

刘 艺 著



前 言

《Delphi 模式编程》经过多年的酝酿和一年多的艰难写作终于完稿了。这本书最初仅仅是我本人学习模式的一个私人笔记，所以使用《Thinking in Patterns with Delphi》的英文名称可能更为准确。

本书第一部分“模式编程原理”，阐述了模式的概念，深入讨论了 Delphi 的模式编程机制和模式编程法则；后面各部分则分别围绕 23 个经典的设计模式进行解说，介绍其结构和用法，并给出模式编程的实践范例。

对于有编程经验的 Delphi 程序员来说，阅读这本书并不困难。书中的例子他们大都很熟悉，只不过以前可能没有用模式编程的方式实现过。在比较不同的思考问题的方法和解决问题的途径后，我相信他们会有“于我心有戚戚焉”的感受。在理解模式的基础上，如果进一步深入阅读本书的“Delphi 的模式编程机制”和“模式编程法则”两章将会有更高层次的收获。

诚恳地讲，这部书并不适合初学者阅读，甚至不适合没有建立面向对象概念和不了解面向对象编程的读者阅读。不过初学者可以先积累一些编程实践经验，并通过阅读本人的《Delphi 面向对象编程思想》来建立面向对象的思维方式。然后，尝试阅读本书的一些较为简单和常用的模式，例如：Factory Method 模式、Strategy 模式等。

本书的结构是松散的，各个模式相对独立，自成一章。强烈建议读者在阅读时，先跳过那些你们认为难读的章节和暂时用不上的模式。我并不是说这些章节不重要，而是说最后再回过头来阅读这些章节效果会更好！

当然也可以将此书作为一本模式编程参考手册，便于读者在项目开发中遇到实际的设计问题时直接查阅相关章节，而不需阅读全书。

本书的光盘中包含了书中绝大多数示例程序的源代码，并在 Delphi7 上调试通过。

本书的其它相关资源和技术支持，可以在我的个人网站和博艺论坛上获得：http://www.liu-yi.net 。另外，感兴趣的个人和单位亦可直接和我本人联系相关的培训。

由于本人水平有限，加之可能的打字笔误，书中难免会有疏漏之处。为此我在博艺论坛（http://www.liu-yi.net/bbs/index.asp）上开辟了《Delphi 模式编程》讨论版，欢迎大家及时把勘误意见贴在上面，以便在重印时修订。

最后要感谢邵印中为本书所做的校对工作，感谢周赛锋为本书提出了很好的建议，感谢段立、罗宾、李启元、洪蕾、吴永逸、吴英在本书写作中给予的支持。如果没有家人、朋友、读者的厚爱，本书可能永远无法完成。

还要衷心感谢多年来不断支持我技术写作的机械工业出版社华章公司，与他们合作是令人愉快的！同样他们在计算机图书出版界的成绩也是有目共睹的。

刘 艺

www.liu-yi.net

2004 年 7 月 5 日于南京

序

“Design patterns help you learn from others' successes instead of your own failures.”

——Mark Johnson

软件开发是一项极具挑战性的工作

《设计模式》[1]的作者在该书的开篇感叹道:“设计面向对象软件比较困难，而设计可复用的面向对象软件就更加困难”。的确，软件开发是一项极具挑战性的工作。对于编程人员而言，要做出一个良好的设计往往需要经过数次探索和反复尝试，并在大量的经验和教训之中才能找到一个较好的解决方案。尽管以编程为艺术的执着追求者们力求使面向对象设计更加灵活、优雅、健壮，但为这一目标所付出的辛苦代价同样惊人。

好在软件业经过多年的发展已经从崇尚个人软件英雄的手工劳作时代进入了以软件工程为指导崇尚团队合作的软件大生产的工业时代。这就是说，通过集体的贡献，一个系统可以由一些可复用的软件实体（例如：组件、类等）来架构，而无需一切从头开始。

同样，在软件设计中也是如此，通过设计模式，我们可以针对一些特定的问题和场景使用一些现成的固定的模式，而不必为找到一个好的方案做重新的探索。模式，实际上就是前人积累的一些宝贵经验的抽象和升华。这些宝贵的经验得以用文字等有效的形式记录下来，为我们学习和应用提供了极大的方便。

只要是一再重复出现的事物，就可能存在某种模式

所谓模式，简单地说就是从不断重复出现的事物中，发现和抽象出的规律，是解决问题经验的总结。只要是一再重复出现的事物，就可能存在某种模式。例如：小桥流水、曲径通幽的中国园林模式；柳眉杏眼、巧笑倩兮的古典美女模式；枕山环水、背水面街的吉祥风水模式；飙车枪战、美女英雄的 007 电影模式；花言巧语、死缠烂打的泡妞追女模式。

备受模式社区推崇的建筑学家 Christopher Alexander 说：“每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动”。尽管他所指的是城市和建筑模式，但他的思想也同样适用于软件模式，只是在面向对象编程的解决方案里，我们用对象和接口代替了墙壁和门窗。两类模式的核心都在于提供了相关问题的解决方案。

[1]《Design Patterns：Elements of Reusable Object-Oriented Software》，中译本《设计模式：可复用面向对象软件的基础》已由机械工业出版社出版。文献引用中经常称该书的 4 位作者 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 为 GoF，即“四人帮”。本书中也沿用 GoF 这种约定成俗的称谓。

大凡讲述软件模式的文章和书籍，总少不了把模式的源头追溯到 Christopher Alexander，广大软件开发者也因此熟悉了这位模式思想的先驱，许多人逐渐成为他的思想信徒，并把他的《建筑的永恒之道》和《模式语言》奉为经典。

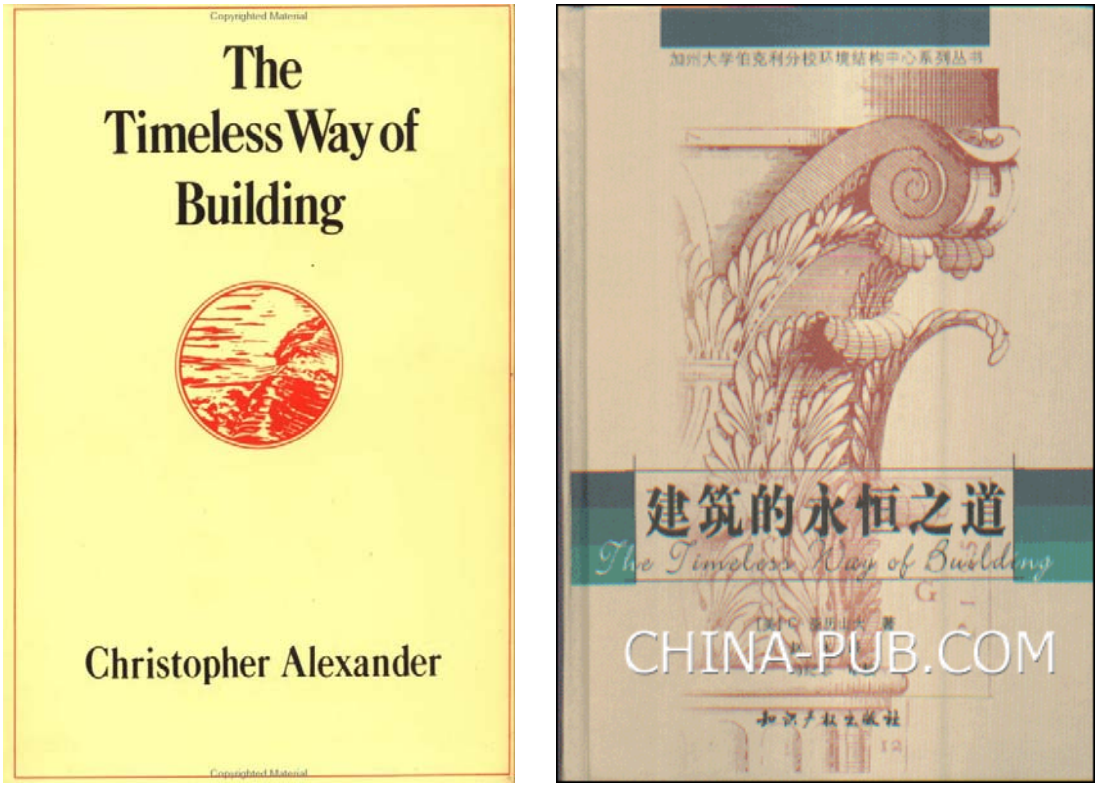


图 1Christopher Alexander 的《建筑的永恒之道》

模式的源头在中国

其实，最早把模式的思想应用于城市和建筑中的不是 Christopher Alexander，而是中国人。模式的源头应该在中国！

早在 2 千多年以前，中国就形成了自己的城市建设模式，《周礼 • 冬官考工记第六》里记载有古代都城建设模式：“匠人营国，方九里，旁三门，国中九经九纬，经涂九轨，左祖右社，面朝后市，市朝一夫。”就是说，城市应该呈方形，每边九里，四边城墙上各设有三座城门，城内有九条直街与九条横街(或者理解为横竖各为三条街，而每条街都由三条并列 的道路组成)，街道之宽为车轨的九倍。城市中前面为朝廷部分（政治中心），后面为商市部分（商业中心），朝、市每边均为百步之宽。城市的左方有祖庙，右方为社稷坛。这种方整有序的城市规划一直为中国历代的封建王朝所依循并得到进一步的发展，我们从汉末三国时期的邺城、唐长安、宋汴梁一直到明清的北京，都可以见到这种城市模式。

北宋时期，李诫编修的我国古代建筑学经典著作《营造法式》体现了中国人在建筑模式方面的智慧，该书 1068 年开始编修，1100 年成书，1103 年刊行，历时 30 多年。比 Christopher Alexander 的《建筑的永恒之道》要早 900 年。全书共 34 卷，其内容来自熟练工匠的经验，总结了官式建筑的模式和规范。

十分有趣的是 GoF 在《设计模式》一书中归纳出模式的四个基本要素为：模式名称（pattern name）、问题（problem）、解决方案（solution）、效果

（consequences）；而李诫的《营造法式》则分为释名、各作制度、功限、料例和图样 5 部分。通过对比我们不难发现，其中释名相当于“模式名称”，给出模式的定义，便于交流和记忆。各作制度相当于“问题”，描述了应该在何时使用模式。它解释了设计问题和问题存在的前因后果，它可能描述了特定的设计问题。功限相当于“效果”，描述了模式应用的功效及使用模式应权衡的问题。料例和图样相当于“解决方案”，是解决所阐述问题的一个构造或配置。Christopher Alexander 的强调“模式是某种场景下某个问题的解决方案”，所以这种解决方案不是抽象的，必须结合具体的料例和图样。料例和图样相当于《设计模式》中的代码示例和模式的 UML 结构图。

模式的思维方法更符合中国式的智慧

由此可见，中国人早在古代就已经掌握了认识和应用模式的方法。模式的思维方法更符合中国式的智慧。

图 2 李诫编修的《营造法式》

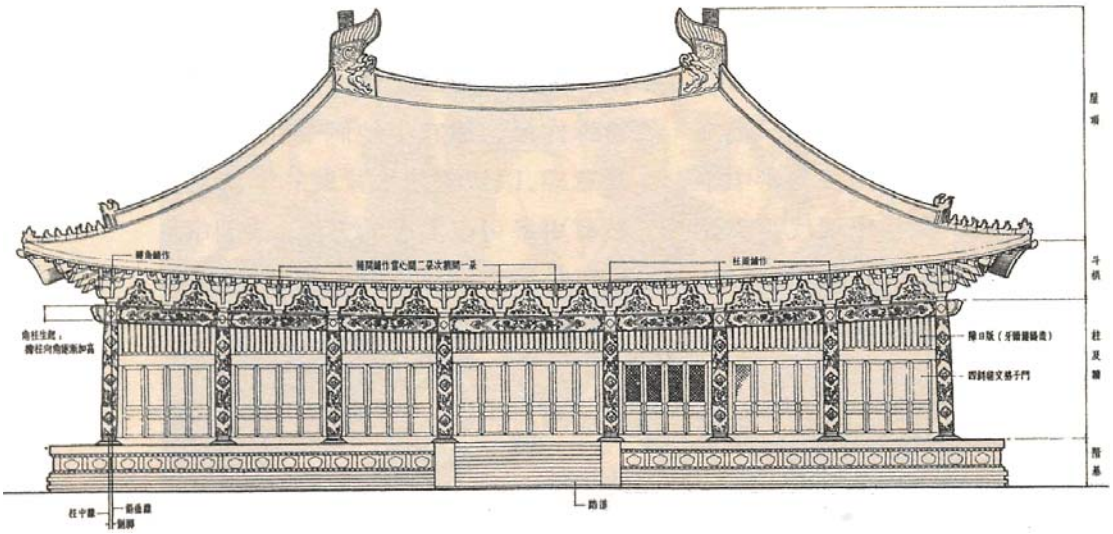


图 3《营造法式》中的立面处理图样

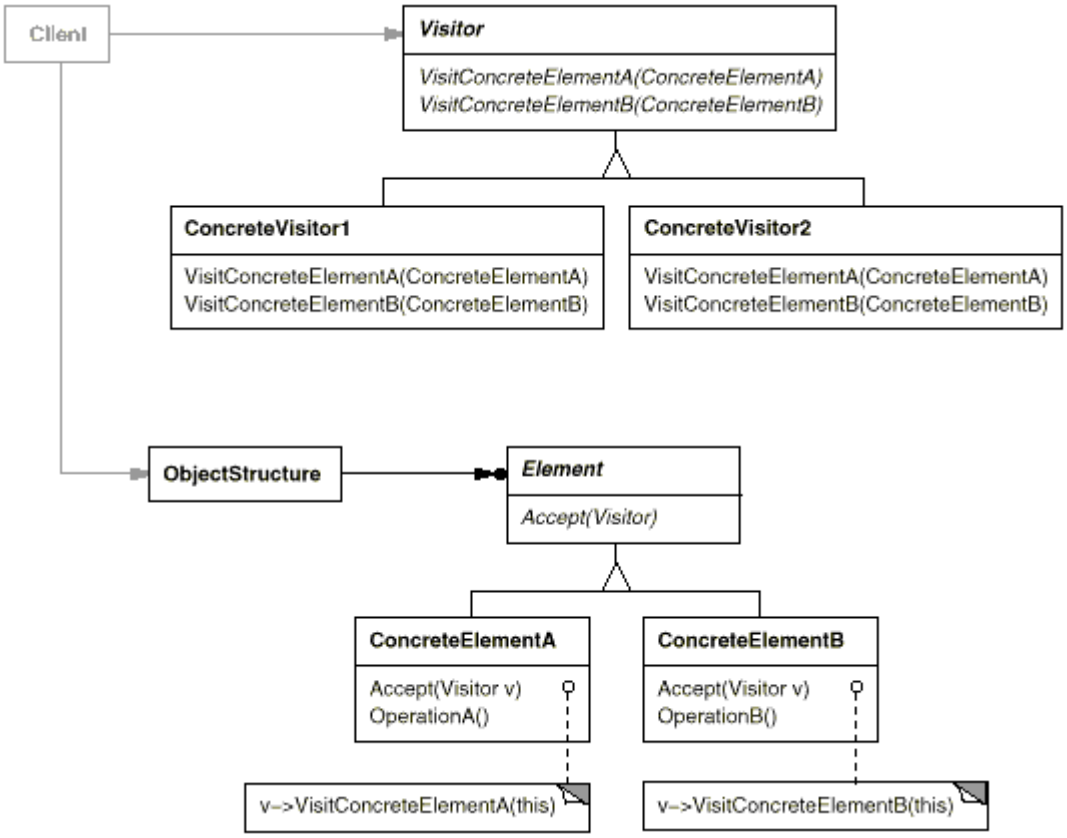


图 4《设计模式》中的 Visitor 模式结构图

虽然模式的概念早就有了，但要上升到实用阶段，成为可以定义并广泛接受的模式并不简单。模式的核心就是特定的解决方案，它有效且有足够的通用性，能解决重复出现的问题。模式的另一种视角是把它看成是一组建议，而创造模式的艺术则是将很多建议分解开来，形成相互独立的组，在此基础上可以相对独立地讨论它们。

模式的关键在于其源于实践并指导实践

模式的关键在于其源于实践并指导实践。所以要发现模式，必须观察人们的工作过程，发现其中优秀的设计经验，并找出“这些解决方案的核心”，这并不是一个简单的过程。如果你能发现某个模式，它将是非常有价值的。你不必死背模式，甚至不必通读任何一本有关模式的书，你只需要了解到这些模式都是干什么的、它们解决什么问题、它们是如何解决问题的就足够了。这样，一旦你碰到类似问题，就可以从书中找出对应的模式来解决问题。那时，你再深入了解相应的模式也不迟。所以，很多有丰富开发经验的程序员学习模式时都会有似曾相识、相见恨晚的感觉。他们会说“唉，原来这个问题可以这么解决！”

模式不是什么新鲜概念了。因此，模式并不是那些模式的作者人为“发明”的，而是他们从自己和别人的大量实践经验中“发现”的。他们的职责是记录通用的解决方案，找出其核心，并把最终的模式记录下来，传授给大家。对于一个高级开发人员，模式的价值并不在于它给予你一些新东西，而是在于它能帮助你更好地交流。如果你和你的同事都明白什么是 Facade 模式，你就可以这样非常简捷地交流大量信息：“这个类是一个 Facade 模式。”也可以对新手说：“用抽象工厂模式来解决这个问题。”模式为设计提供了一套词汇，这也是

为什么模式名字这么重要的原因了。

《设计模式》作者之一的 John Vlissides 指出模式带来的四大好处是：

1. 它们记录了专家的经验， 并且让非专家也能理解。

2. 它们的名称构成了一份词汇表， 帮助开发者更好的交流。

3. 它们帮助人们更快的理解一个系统——只要这个系统是用模式的方式描述的。

4. 它们使系统的重组变得更容易， 不管原来的系统是否以模式的方式设计的。

John Vlissides 说：“过去， 我一直认为第一项是模式最大的好处。现在我认识到， 第二条起码也同样重要。请想一想，在一个开发项目的过程中， 有多少字节的信息在开发者之间流动（ 包括口头的和电子的）？我猜，就算没有 1G 也有好几兆。（在推出了《设计模式》之后，我已经收到了好几十兆给 GoF 的电子邮件。而且我们所描述的还都是小型到中型的软件开发项目。）由于有如此之大的信息交流量，所以效率上任何微小的提升都能大量节约时间。在这个意义上， 模式拓宽了人们交流的带宽。我对第三、四条的评价也在逐渐提高， 特别是在项目越来越大、软件生存周期越来越长的今天。 至少在短期内， 模式主要存在于大脑中， 而不存在于工具中。如果有了方法学或自动化工具的支持，应该还有其他的收益， 但是我相信这些都只是蛋糕上的奶油，而不是蛋糕本身， 甚至都不能算蛋糕的一层。”

由此可见，模式并不依赖于工具或方法学的支持，自动化工具也不能使模式的应用更有效。因为模式本身就很灵活，你不能盲目地使用，一旦需要使用模式，你必须知道如何将它运用于当前的问题。这也是关于模式的自动化工具为什么都遭遇失败的原因。与我们经常使用的组件则正好相反，实际上没有哪个模式是能让你不假思索就使用的规则或现成的程序。模式是一种“半成品”，它仅仅给出了的解决方案，但并不告诉你如何去实施。为了用好它，你还必须在自己的项目中把剩下的那一半补上。我本人每次在使用模式时，都在不断地改动。即使同一个解决方案，但没有一次是完全相同的。虽然每个模式相对独立，但又不彼此孤立。有时候它们相互影响，有时互相配合，在具体实践中，我们还可能将模式混合使用。例如在本书中，我在访问者模式一章中就给出了这样一个模式混合使用的范例。

模式并不代表模式的应用。许多自称精通模式的人，他们也只不过是能把 GoF 的 23 种模式背得滚瓜烂熟，并用来开开玩笑而已。我们不难在网上看到诸如：打篮球的模式、泡妞的模式，却很少看到模式在编程上的真正应用。

模式的困难在于模式既是一种解决问题的方法，更是解决问题的本身。

模式应用要比理解模式本身更加困难。模式的困难在于模式既是一种解决问题的方法，更是解决问题的本身。因为针对同一类问题，站在不同的思考角度会有不同的模式选择。例如，解决去耦合的问题，就有用于前台对象和后台对象之间去耦合的 Proxy 模式；用于系统去耦合的 Observer 和 Mediator 模式；用于请求发送者和接收者之间去耦合的 Chain of Responsibility 模式；用于抽象部分和实现部分之间去耦合的 Bridge 模式；等等。同样都是解决去耦合的问题，至于选择哪种模式，如何应用这些模式，就要看你的解决问题的出发点以及对各方面利弊的权衡了。

世界是复杂多变的，软件设计的问题在于明确目标、把握方向是件非常困难的事情。为什么说设计的本质是权衡，就在于你很难只确定一个目标，更难确定一个固定的目标。你要考虑的往往是在多种约束下的多种可能解决方案。

实际上没有哪种解决方案是唯一正确的，这就是设计策略和风格的问题。因此模式也不是万能的，模式有时仅可以看作是给你一个解决问题的建议、一个可供借鉴的经验、一个可以探索的方向。不同境界不同水平的人对模式有不同的理解，应用起来效果也不一样。

模式不是万能的，模式什么都不保证

所以那些认为模式可以保证程序更加简洁、优雅和健壮，模式可以保证软件更好的复用性的想法是幼稚可笑的。道理很简单， 模式什么都不保证，模式无法取代人的位置。那些对模式走火入魔的人，往往对更加简单的代码编写方式视而不见，在编程中盲目使用模式。例如，本来可以用简单的条件表达式就足够解决的两三种不同的计算方法，却非要使用 Strategy 模式，反而使程序更加复杂难以理解，无异于用大炮打蚊子。

有极端看法的人甚至认为：“水平没到，学也白学，水平到了，无师自通。所以不要学设计模式。”

这种说法虽然有一定的道理，但我并不完全同意。

模式对于初学者虽然很难，但了解和学习模式肯定比不知道模式好。尽管他们都会经过一个迷信模式、痴迷模式、滥用模式的阶段，但这不可怕，因为有招总比无招好。只要掌握正确的学习方法，活学活用，肯定要比无师自通的人少走弯路。而且随着不断的实践，在模式应用上也会更加自如。

在实践和思考中学习模式

模式正确的学习方法就是在实践和思考中学习模式，Jim Coplien 在《Software Patterns》一书中，写道：

“我更愿意用剪裁衣服的模式为这个定义打比方。我可以通过在布料上以剪裁的角度和长度来定义剪子的路径，以此告诉你如何裁一件衣服。或者，我也可以给你一个模式。你阅读剪裁说明，你可能根本就不知道在做什么，但是如果你以前做过类似衣服，你就能完成。模式预示着产品：它既是制造产品的规则，在很多方面它又是产品本身。”[2]

所以要学习模式，首先要有编程实践经验。对于没有编程实践经验的初学者并没有必要急于学习模式。在初学者获得一定编程实践经验之后，我认为应该让他们先掌握几种常用的模式作为解决方案，然后去努力的在现实中寻找使用这些方案的合适场景。学习模式不要指望能够一口气全部学完所有的 23 种设计模式。即使你博闻强记，能够学完，如果比较少用，一阵子之后也可能会忘记。通常建议你通读模式书籍时能先大概了解这些模式，并根据自己的编程实践有针对性的选择那些感兴趣的模式进行实践探索。等碰到类似问题，在从书中找出对应的模式来解决问题，深入了解相应的模式也不迟。

近几年来“ 模式” 这个词真是非常流行。就像任何流行的东西一样，GoF 的经典著作《设计模式》也一时洛阳纸贵，备受推崇。现在模式对于软件开发的影响越来越大，国内开发人员对模式的了解虽然比国外的开发人员晚了数年的时间，但能如此引起重视的确是一件好事。不过我敢肯定，GoF 的《设

[2] 原文如下：

I like to relate this definition to dress patterns. I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself.

《设计模式》并不是一本可以轻松阅读的书

计模式》并不是一本可以轻松阅读的书。这是因为《设计模式》整理自 Erich 的博士论文，其论文大约占了整部书的一半。对于程序员或普通读者而言，这本书过强的学术性语言和过于严谨的论述无疑增加了阅读的难度，使得这本薄薄的经典如同阳春白雪，曲高和寡。有程序员甚至抱怨自己用一个多月的时间也啃不下这本两百多页（中译本）的薄薄小书。这也就是为什么会有那么多解释这本经典的书出现的原因。

中国人向来有注疏经典的优良传统。以《周易》（易经）为例，就有朱熹的《周易本义》、王弼的《周易正义》、王夫之的《周易稗疏》、苏轼的《东坡易传》等数百种之多。同样，像《设计模式》这样的软件业的经典著作，通过“注疏”来学习并没有什么不好。既然市面上已经有了《Java 与模式》、《Visual Basic 设计模式》这样的书籍，为什么不能有一本属于 Delphi 的模式书籍？

准确地讲，与 Delphi 有关的模式书籍我们也能找到一些。最早见到的是 ModelMaker 的《Design Patterns Reference》，里面零星介绍了一些用于 Delphi 的设计模式，这些模式仅仅当作 ModelMaker 的商业卖点介绍，而且语焉不详。然后是网上广为流传的一本《设计模式》的 Delphi 版电子书,这本书是台湾人把《设计模式》中 C＋＋的例子翻译成 Delphi 的翻版，令人遗憾的是这种生搬硬套不但使得 Delphi 改写的程序难以运行，甚至还存有不同语言之间理解上的谬误。不久前看到李维的《Inside VCL》，但是并不像广告宣传的那样有多少与模式有关的内容，该书仅仅提到了经典模式中的 2 个，即 Facade 和 Command 模式。而在另一本国内作者陈省的《Delphi 深度探索 第 2 版》中，虽然有一章专门介绍设计模式，但是 80 页的篇幅似乎过于简洁，而且仅限于“以 VCL 中的示例来介绍如何使用 Delphi 应用设计模式的内容”。

应该有一本能够帮助 Delphi 程序员比较全面理解和学习模式的书籍

我觉得应该有一本能够帮助 Delphi 程序员比较全面理解和学习模式的书籍。模式的困难在于其很强的实践性，所以写这样一本书有很大的难度。我写这本书的目的并不是为了在 GoF 的模式上做什么创新，而是为了通过通俗易懂的解说和明白实用的例子来帮助 Delphi 程序员掌握模式的使用，建立模式的思维。因此，我花费了大量的时间和精力改编和设计了实践模式的 Delphi 编程范例，其目的是为了让读者看到模式在实践应用中的价值。

可能一些程序员在阅读技术书籍时会固执地说：“别给我们讲大道理，给我们看程序!”

我要说他们的这些要求是对的！很多关于模式的书故作高深，晦涩难懂，其示例代码也是凤毛麟角，让我们看不到任何可以执行的完整代码。我反对这种风格的技术写作。如果我们的目的不是传授制知识，解决问题，又为何要浪费笔墨？

用最通俗易懂的语言和最明白实用的范例解说深奥睿智的《设计模式》

我的这本《Delphi 模式编程》（《Thinking in Patterns with Delphi》）可以看成是 Delphi 程序员的模式入门和实践读物。我的想法是力求用最通俗易懂的语言和最明白实用的范例解说深奥睿智的《设计模式》。

为了让更多的人看懂这本书，我在每个模式的开篇都会用生活中的例子和图示进行形象的模式解说；为了便于套用模式，我为每个模式提供了 Delphi 的代码模板；为了给程序员看程序，我为每个模式精心准备了最实用的范例程序，附上完整的源代码和代码剖析，保证你能运行这些真正的模式应用程序（而不是一个空壳）。这些范例程序涉及行业应用领域包括：酒店管理系统、银行管理系统（转账系统、信用卡管理系统）、人力资源管理系统（组织机构管理、薪酬福利管理）、地理信息系统、聊天室系统、任务调度系统、项目审批系统

等。

当然，书上的范例毕竟只是经过处理的简单的例子，甚至还会有为讲解模式而模式之嫌，离特殊而复杂的实际应用还有距离。就像模式什么都不保证一样，范例程序同样不能保证可以复制到你的项目中取代你自己编程，因此不要指望有太多现成的代码可以拷贝。这本书只是帮你理解设计模式，告诉你这个模式是怎么一回事，大概可以处理什么样的情况，这样处理有什么好处，以及如何用 Delphi 编程实现。最终的解决方案需要你自己去权衡，模式编程需要你自己去完成。

模式只是开始，并非终点

当你使用模式时请记住，它们只是开始，并非终点。让我们这些作者去囊括项目开发中的所有变化和技术是不可能的。我编写《Delphi 模式编程》的目的也只是作为一个开始，希望它能够把我自己的和我所了解的经验与教训传递给读者，你们可以在此基础上继续努力。请读者记住的是，所有模式都是不完备的，如果你们决定进入模式编程的新境界，你们就有责任和兴趣在自己的软件开发中实践它们、完善它们。

刘 艺

www.liu-yi.net

2004 年 7 月 2 日于南京

目 录

第一部分　模式编程原理

第1章　模式概述

1.1　模式的概念

1.1.1　什么是模式

模式（Pattern）的概念最早由建筑大师 Christopher Alexander 于二十世纪七十年代提出，应用于建筑领域。

八十年代中期由 Ward Cunningham 和 Kent Beck 将其思想引入到软件领域，1994 年开始由 Hillside Group（由 Kent Beck 等发起成立）和 OOPSLA 联合发起了国际程序设计模式语言大会（Patterns Languages of Program Design，简称 PLoP 或 PLoPD），如今模式已成为软件工程领域内的一个热门话题，其在计算机领域的影响超过了在建筑界的影响。

软件业界的模式热最早源自 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides[1]的经典著作《Design Patterns：Elements of Reusable Object-Oriented Software》[2]。这本书用 C++和 Smalltalk 语言的示例代码来阐明了 23 种设计模式及其实现。模式的推广和应用近年来发展十分迅速，涉及程序设计、系统架构、教学研究、组织管理等多个方面。在推动模式中起重要影响的著作还有 Frank Buschmann、Regine Meunier、Hans Rohnert、Peter Sommerlad 和 Michael Sta[3]合著的《Pattern-Oriented Software Architecture：A System of Patterns》[4]以及程序设计模式语言大会的精选论文集《Pattern Languages of Program Design》和《Pattern Languages of Program Design 2》

Christopher Alexander 说过：“每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动”。尽管他所指的是城市和建筑模式，但他的思想也同样适用于面向对象设计模式，只是在面向对象的解决方案里，我们用对象和接口代替了墙壁和门窗。两类模式的核心都在于提供了相关问题的解决方案。

Christopher Alexander 将模式分为三个部分：

- **Context——**场景（也称为上下文），指模式在何种状况下发生作用
- **System of Forces——**动机，意指问题或预期的目标；
- **Solution**——解决方案，指平衡各动机或解决所阐述问题的一个构造或配置。

他提出，模式是表示场景、动机、解决方案三个方面关系的一个规则，每个模式描述了一个在某种场景下不断重复发生的问题，以及该问题解决方案的核心所在，模式即是一个事物（thing）又是一个过程（process），不仅描述该事物本身，而且提出了通过怎样的过程来

[1] 文献引用中经常称这 4 人为 GoF，即“四人帮”。本书中也采用 GoF 这种约定成俗的称谓。
[2] 中译本《设计模式：可复用面向对象软件的基础》已由机械工业出版社出版。
[3] 文献引用中经常称这 5 人为 GoV，即“五人帮”。
[4] 中译本《面向模式的软件体系结构 卷 1：模式系统》已由机械工业出版社出版。

产生该事物。这一定义已被软件界广为接受。[1]

《Understanding and Using Patterns in Software Development》一书的作者 Dirk Riehle 和 Heinz Zullighoven 给出了“模式”一个最宽泛的实用定义：

“模式是不断重现的具体之抽象，这种重现发生在特定的而非任意的场景中。”[2]

该书作者指出，在软件模式的社区中，模式的观念与在设计中如何解决问题密切相关。更明确的是，不断重现的具体形式为重复出现的问题提供了可行的解决方案。但是模式不仅仅是解决重复发生问题的实战经验，虽然问题总在某一特定的场景中发生，但却呈现众多的相关动因或动机，而推荐的解决方案包括某些能够平衡这些动因的结构，该结构的形式适合给定场景。使用模式的形式，即这种解决方案的描述，试图洞悉并抓住其体现出的本质，以便其他人从中学习，用于相似情况中。模式也是一个给定的名称，作为一个概念上的句柄，以便该模式及其代表的那些有价值的信息便于讨论。

在模式社区关于模式更通俗易懂的定义是：

“模式好比是充满真知灼见的命名了的金块，它是被验证的解决方案的精华所在，该方案解决与某个计算有关的场景中重复发生的问题。”[3]

许多人根据 Christopher Alexande 的定义，仅仅看到“模式是某种场景下某个问题的解决方案”，这是远远不够的。《设计模式》作者之一的 John Vlissides 把它称为“误解”，并举过一个有趣的反例进行批驳：

- 问题：我获奖的奖券就快过期了，我怎样拿到奖金？
- 场景：离截止时间只有一小时，可是我家的小狗把奖券给吞了。
- 解决方案： 把狗开膛破肚，掏出奖券，然后飞奔到最近的兑奖地点。

这是“某种场景下某个问题的解决方案”， 但它不是模式。还缺了什么？ 至少缺三样东西：

1. 可重复性。解决方案应该对应于外部的场景。

2. 可传授性。一个解决方案应该可以移植到问题的不同情况上。（ 绝大多数模式的可传授性都建立在“ 约束” 和“ 效果” 的基础上。）

3. 用来表示这个模式的名称。

所以对于模式很难找到一个令人满意的定义。这是因为模式既是一个事物，也是对类似事物的描述。要区分这两者，有一种办法，即让“模式”这个术语只用来指代模式的描述，同时用“模式实例”来指代模式的具体应用。无论如何，任何对模式要素的规定，除了必须包括问题、解决方案和场景之外，都必须提及可重复性、可传授性和名称。

另外对模式的另一大误解是把模式看作是例程、规则、编程技巧、数据结构……。

与我们经常使用的组件则正好相反，实际上没有哪个模式是能让你不假思索就使用的规则或现成的程序，同时模式也不仅仅是“编程技巧”，尽管模式中的一些特定的实现技巧是

---

[1] 参见《The Timeless Way of Building》P.247 （牛津大学出版社 1977 年出版，中文译名《建筑永恒之道》），其关键论述原文如下：
“Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.
As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.
As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.
The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing .”

[2] 原文“ A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts. ”

[3] 原文“ A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns. ”

特定于语言的。但看重“技巧”会导致过分强调解决方案，从而忽视了问题、场景和名称的重要性。

GoF 在《设计模式》一书中归纳出模式的四个基本要素：

- **模式名称（pattern name）** 一个助记名，它用一两个词来描述模式的问题、解决方案和效果。命名一个新的模式增加了我们的设计词汇。设计模式允许我们在较高的抽象层次上进行设计。基于一个模式词汇表，我们自己以及同事之间就可以讨论模式并在编写文档时使用它们。模式名可以帮助我们思考，便于我们与其他人交流设计思想及设计结果。找到恰当的模式名也是我们设计模式编目工作的难点之一。
- **问题（problem）** 描述了应该在何时使用模式。它解释了设计问题和问题存在的前因后果，它可能描述了特定的设计问题，如怎样用对象表示算法等。也可能描述了导致不灵活设计的类或对象结构。有时候，问题部分会包括使用模式必须满足的一系列先决条件。
- **解决方案（solution）** 描述了设计的组成成分，它们之间的相互关系及各自的职责和协作方式。因为模式就像一个模板，可应用于多种不同场合，所以解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合（类或对象组合）来解决这个问题。
- **效果（consequences）** 描述了模式应用的效果及使用模式应权衡的问题。尽管我们描述设计决策时，并不总提到模式效果，但它们对于评价设计选择和理解使用模式的代价及好处具有重要意义。软件效果大多关注对时间和空间的衡量，它们也表述了语言和实现问题。因为复用是面向对象设计的要素之一，所以模式效果包括它对系统的灵活性、扩充性或可移植性的影响，显式地列出这些效果对理解和评价这些模式很有帮助。

对于模式的学习和研究，我们可以从模式的四个基本要素入手，全面理解模式的概念，这对我们掌握模式和应用模式至关重要。

1.1.2 模式可以做什么

凡有成功经验的编程高手都知道：不是解决任何问题都要从头做起。他们更愿意复用以前使用过的解决方案。当找到一个好的解决方案，他们会一遍又一遍地使用。这些经验是他们成为内行高手的重要原因。通过模式解决特定的设计问题，使面向对象设计更灵活、优雅，最终复用性更好。模式能够帮助设计者将新的设计建立在以往工作的基础上，复用以往成功的设计方案。一个熟悉模式的开发者不需要再去发现它们，而能够立即将它们应用于设计问题中。

模式的应用对软件开发所起的重大作用主要在于：

- 模式是人们在长期的设计软件、管理组织软件开发等实践中大量经验的提炼和抽象，是复用软件设计方法、过程管理经验的有力工具。模式类似于武功中的招式，它提供了一系列软件开发中的思维套路。如，通过模式的使用，有利于在复杂的系统中产生简洁、精巧的设计。
- 模式为我们提供了一套简洁通用的设计、管理、组织方面的词汇，同时模式也为我们提供了一个描述抽象事物的规范标准，可大大促进软件开发过程中人与人之间的交流，而软件开发中的交流是至关重要的，“软件项目失败的原因最终都可追溯到信息没有及时准确地传递到应该接收它的人”。

模式关注在特定设计场景中出现的重复出现的设计问题，并为它提供一个解决方案。模

式提炼并提供一种手段来重用从经验实践中获得的设计知识。熟悉模式的开发人员能立即把模式应用到设计问题中而不需要重新发现它们。模式不再使这类知识仅存在于少数专家的大脑中，模式使得这类知识更容易获得。你可以使用这样的专家知识为一个特定任务设计高质量软件。

模式明确并指明处于单个类和实例层次或组件层次之上的抽象。典型情况下，一个模式描述几个组件、类或对象，并详细说明它们的职责和关系以及它们之间的合作。所有的组件共同解决模式关注的问题，而且通常比单个组件更有效。

模式为设计原则提供一种公共的词汇和理解。模式名称，如果仔细选择，会成为广泛传播的设计语言的一部分。它有助于设计问题及其解决方案的有效讨论。你可以使用一个模式名称，解释解决方案的哪个部分对应模式的哪个组件，或者对应它们之间的哪个关系。从而避免了用冗长而复杂的描述来说明针对某些特殊问题的解决方案。

模式提供一个功能行为的基本骨架，从而有助于实现应用程序的功能。另外，模式清楚描述了软件系统的非功能需求，如可更改性、可靠性、可测试性或可重用性。

模式有助于建立一个复杂的架构。每个模式提供组件、作用以及相互关系的预定义集。它可以用于指定具体软件结构的特定方面。这种使用预定义设计人工制品的方法提高了设计的速度和质量。理解并应用现成的模式比起你寻找自己方案来要节省时间。这并不是说现有模式一定会比你自己的方案好，但是，至少如本书所介绍的模式有助于你评价和评估设计方案。

目前不少程序员对模式能做什么要么夸大其词的，要么心存疑虑，这都是因为对模式的作用缺乏正确的认识。

《设计模式》一书作者，GoF 之一的 John Vlissides 说过：“ 在创造新事物的过程中， 模式无法取代人的位置。它们只是带来一种希望， 有可能让一个缺乏经验的、甚至是未入门的，但是有能力、有创造性的人尽快获得设计的能力。”这是目前为止对模式的作用最客观、最冷静的评论。

而 Mark Johnson 说得更好：“ 设计模式帮助你从别人的成功中而不是自己的失败中学到东西。”[1]许多模式的受益者都有这种体会。

1.2 模式与架构

1.2.1 什么是架构

架构（Architecture），也称为软件体系结构，用来指可以预制和可重构的软件框架结构。架构尚处在发展期，对于其定义，学术界尚未形成一个统一的意见，而不同角度的视点也会造成软件体系结构的不同理解。

ANSI/IEEE 610.12-1990 软件工程标准词汇对于体系结构定义是：“ 体系架构是以构件、构件之间的关系、构件与环境之间的关系为内容的某一系统的基本组织结构以及知道上述内容设计与演化的原理 ”。

Mary Shaw 和 David Garlan 认为软件体系结构是软件设计过程中，超越计算中的算法设计和数据结构设计的一个层次。体系结构问题包括各个方面的组织和全局控制结构，通信协议、同步，数据存储，给设计元素分配特定功能，设计元素的组织，规模和性能，在各设计方案之间进行选择。Garlan & Shaw 模型的公式是：

[1] 原文：“ Design patterns help you learn from others' successes instead of your own failures. ”

```
Architecture ={component, connector, constrain}
```

其中组件（component）可以是一组代码，如程序的模块；也可以是一个独立的程序，如数据库服务器。连接件（connector）可以是过程调用、管道、远程过程调用（RPC）等，用于表示组件之间的相互作用。约束（constrain）一般为对象连接时的规则，或指明组件连接的形式和条件，例如，上层组件可请求下层组件的服务，反之不行；两对象不得递规地发送消息；代码复制迁移的一致性约束；什么条件下此种连接无效等。

1.2.2 架构和模式的关系

因为架构和模式在当前的软件开发中经常地被提及，可是很多人容易混淆这两个术语，而对此，学术界也没有一个非常统一的定义。

《Pattern-Oriented Software Architecture：A System of Patterns》从广义上将模式分为三类，每一类都由具有类似范畴度量或抽象的模式组成：

- **架构模式（Architectural Patterns）** 架构模式表示了软件系统的基本结构化组织纲要。它提供一套预先定义好的子系统，规定它们的职责，并包含用于组织它们之间关系的规则和方针。架构模式可作为具体软件体系结构的模板。它们规定一个应用系统范围的结构特性，并对其子系统上的体系结构施加影响。所以架构模式的选择是开发一个软件系统时的基本设计决策。
- **设计模式（Design Patterns）** 设计模式提供了一个用于细化软件系统的子系统或组件，或它们之间的关系的纲要。它描述了通信组件的公共再现结构，通信组件是在特定语境中一般设计问题的。设计模式是中等规模的模式。在规模上它们比架构模式小，但又独立于特定编程语言或编程范型。设计模式的应用对软件系统的基础结构没有影响，但可能对子系统的体系结构有较大影响
- **惯用法（idiom）** 惯用法是对应于一种编程语言的低层模式。惯用法描述了如何使用给定语言的特征来实现组件的特殊方面或它们之间的关系。惯用法代表了最低层模式。它们关注设计和实现方面。大多数惯用法是依赖语言的——它们把握现有的编程经验。同一个惯用法对不同的语言常常看起来不一样，有时候一种惯用法对一种编程语言有用而对另一种语言却无意义。

实际上，架构和模式应该是一个属于相互涵盖的过程，但是总体来说架构更加关注的是所谓的高层设计，而模式关注的重点在于通过经验提取的准则或解决方案在设计中的应用，因此在不同层面考虑问题的时候就形成了不同问题域上的模式。相对于系统分析而言，架构是在提出解决问题的方案，而系统分析则是解决这些问题，这两者都会运用到模式，只是侧重的角度略有不同，架构方面倾向于架构模式，而系统分析更多的是采用设计模式和具体语言的实现模式。架构强调的是软件系统的结构及其各个元素之间的关系，而模式则是抽象各个层次上的关系，比如在架构设计时，设计模式是经过经验抽象过后的一些“准则”，而利用这些模式，有助于在架构设计中更好的分离系统元素和组织系统元素之间的关系。

模式是一个经验提取的“准则”，并且在一次一次的实践中得到验证，在不同的层次有不同的模式，小到语言实现（如 Singleton 模式）大到架构（如 Facade 模式）。在不同的层面上，模式提供不同层面的指导，比如架构设计方面，三层应用程序，分布式应用程序等等这些技术架构模式为架构设计提供了理论的参考，而在程序设计领域，设计模式则是提供了描述各个元素（在面向对象领域，更多的是指 Class）之间的关系。

相对于系统分析或者设计模式来说，体系结构从更高的层面去考虑问题，所以关注的问题就体现在“不变”因素上，比如系统部署中，更加关心应用程序的分层分级设计，而在这个基础之上提出的部署方案，才是架构考虑的重点。体系结构关心应用程序模式，更加体现在通过技术去解决这些业务差异带来的影响，关心是否是分布式应用程序，关心系统分层是如何设计，也关心性能和安全，因此在这样的情况之下，会考虑集群，负载平衡，故障迁移等等一系列技术。

希望通过定义的方式来区分架构和模式是不太可能的，因为本来就是交互交叉和提供服务的，比如 Proxy 模式就是一个例子，在设计模式中是一个非常经典的模式，在架构中同样适用。对于熟悉架构设计的系统架构师而言，似乎可以用如下来解释架构和模式之间的关系：架构是高层设计，着眼于不同业务中共性的解决方案，而模式是通用原理。模式用来指导架构设计，同时架构设计选择模式

1.3 从面向对象编程到模式编程

1.3.1 关于封装的哲学

什么东西最好用，显然是简单的东西。然而现实中，很多东西并不简单，但仍然很好用。这是因为设计者封装了复杂性，简化了操作性。例如图 1-1 所示的自动柜员机（ATM），你只要按一下按钮就可以从银行卡上取款，实在是方便。但是，其内部的构造却异常复杂。好在复杂的元器件和线路被封装在一个埋在墙中的黑盒子里，客户只需按一下按钮而无须知道其内部构造和运行机制。

从图 1-1 中，我们可以发现，ATM 面对客户的只是一个按钮，即操作的**接口**，而封装在机器内部的元器件和线路才是具体**实现**。接口是**抽象**的，因为按钮本身并没有实际意义，但通过银行和客户的约定，按钮代表了某个业务，当客户按一下按钮就可以从通过 ATM **具体**实现这个业务，比如取款。客户面对的操作界面是**简单**的和**稳定**的，而 ATM 面对的具体业务实现却是**复杂多变**的。比如新版人民币的出现，就需要为机器增加新的识别功能。但是这种变化必须保证不影响客户的操作。实际上客户的操作没有任何改变，只要按一下按钮就可以从银行卡上取款，无论是新版的还是旧版的，但不会存在金额上的差错。这就是说作为接口的按钮保持了简单和稳定。

这就是封装的哲学！所谓封装就是将具体的、多变的、复杂的实现封装起来，而把简单的、稳定的、抽象的接口留给客户，给客户提供最大的方便。在模式编程中，处处体现着这种哲学。在设计模式中，通常把需要服务的对象称为 Client，本书中会根据上下文称之为客户、客户端或客户对象。设计模式强调客户与服务的脱耦，即服务端的变化不会影响客户端，客户的行为不依赖于某个特定的服务，客户无需知道服务的具体实现。如果还以 ATM 为例，当某个特定的 ATM 出现故障，无法提供服务时，客户可以换一台 ATM，仍然是简单地按一下同样的按钮，就可以获得同样的服务。这就是所谓客户的行为不依赖于某个特定的服务的情形，我们的软件能做到这样吗？换句话说，我们设计的客户对象的行为能够不依赖于某个特定的服务对象吗？

封装的另一个好处就是便于复用。封装通常是多层次的，从封装粒度上讲，封装是针对某一功能和职责的。例如 ATM 内部按照功能和职责可以划分成若干个封装模块，如：货币识别模块、清点计数模块、吐币输出模块等。一个设计良好的可扩展的 ATM，对于新版人民币，只需更换货币识别模块，而无需改动其他模块，充分体现了复用的价值。

模式编程也一样。按照模式编程的单一职责法则（SRP），一个类应该仅有一个原因导

致其变化。这意味着一个类应该只有一个演化方向，一个类只封装一个职责。那么对于整个系统来说，变化才是可控的，因为一个职责上的变动仅涉及到一个类。通常我们把类视为最小粒度上的封装结构。当然，这并不是说复用仅仅是类层次上的，如果设计中在各个层次上都充分体现封装性，那么复用也会存在于各个层次之上。

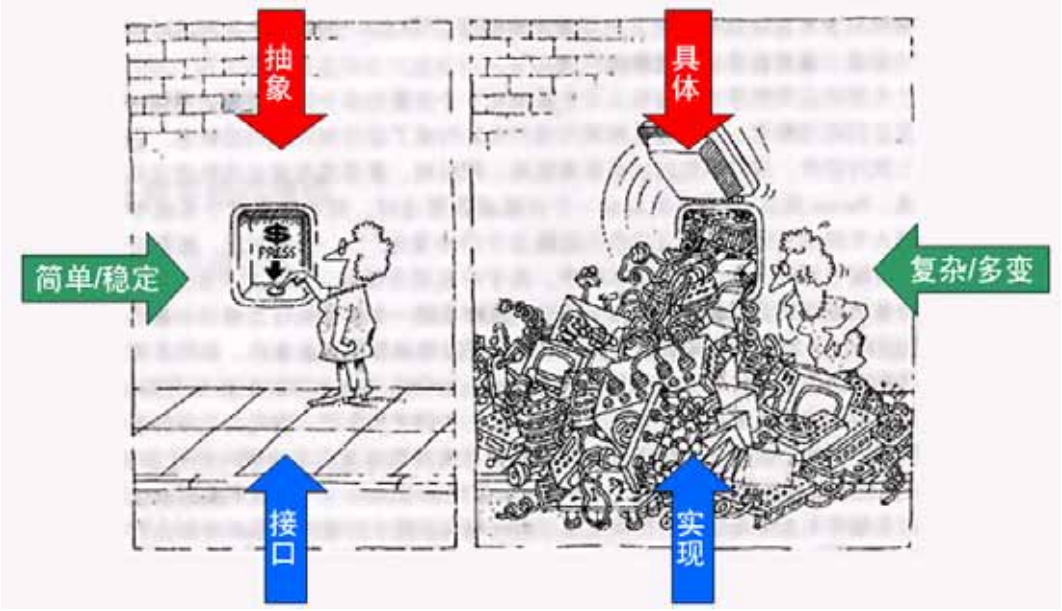


图 1-1 关于封装的哲学

我们通过本小节对封装的理解，可以发现所谓封装就是“对变化的封装”，在系统开发中，我们应该尽量找到系统的可变因素，将其封装，从而使维护变得简单、复用成为可能。这就是 Bruce Eckel 所说的“设计模式的目标在于分离你代码中的变化”。[1]

那么如何实现这种封装，如何实现系统的可维护性和可复用性？我将通过一个面向对象编程向模式编程的演化实例来具体说明。从中我们也能体会到随着编程水平的进化，我们驾驭和解决复杂问题的能力也随之提高。

1.3.2 利用继承实现变化的封装和简单的复用

当许多程序员放弃了面向过程的传统思维方式，开始尝试用面向对象的思想编程时，他们实现软件复用的第一步往往是利用继承来使类之间共享代码，并试图封装变化。

对象的继承是一种在保持对象差异性的同时共享对象相似性的复用。该机制允许类之间共享代码，大大减少了代码长度并且使软件易于维护。对象通过继承，保证了实现部分紧内聚和松耦合的良好特性。

[1] So the goal of design patterns is to isolate changes in your code.

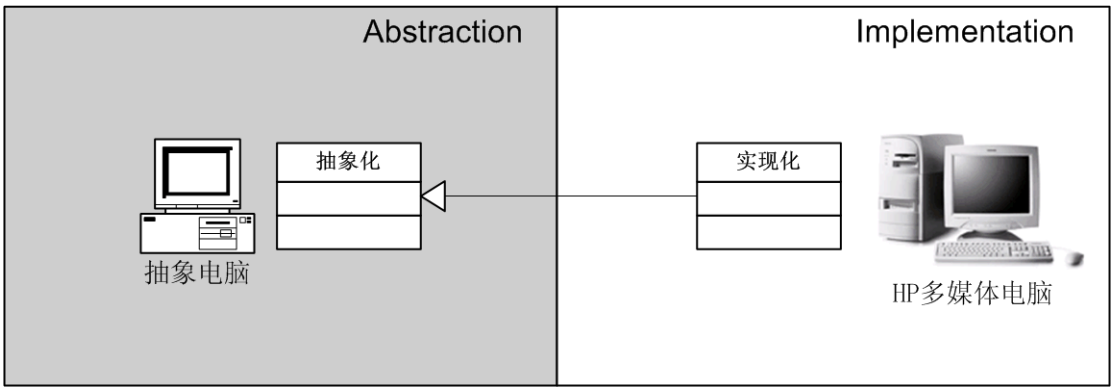


图 1-2 基类与派生类之间的继承关系是一种抽象和具体实现的关系

继承复用是通过扩展一个已有对象的实现来得到新的功能，基类明显地抽象并提取共同的属性和方法，而派生类通过增加新的属性和方法来扩展基类的实现。所以在继承关系中，作为基类的是抽象化部分，这部分用于封装相对稳定和不变的逻辑，作为派生类的是具体化部分，这部分用于封装不易确定和多变的逻辑。基类与派生类的关系应该是抽象接口和具体实现的关系。继承的原则则是从抽象继承而不是从具体继承，而初学者往往会犯滥用继承的错误。后面的章节中，我还会专门进一步讨论继承关系及其正确使用原则。

图 1-2 显示了抽象的电脑和具体的电脑之间的继承关系，它把电脑的抽象化和实现化部分分离开。这种设计是为了提供客户一个稳定和简单的接口，同时还可以满足不同的具体实现。因为客户通常只关心使用电脑解决问题，而不关心具体是什么品牌和型号的电脑、以及电脑是如何工作的。即使客户的需求发生变化，比如客户需要移动办公，具体实现的电脑也可以从台式机换成笔记本或掌上电脑，但客户所需要的电脑功能和操作方式丝毫没有变化，如图 1-3 所示。

一般来说，一个继承结构中的第一层是抽象角色，封装了抽象的业务逻辑，这是系统中不变的部分。第二层是实现角色，封装了设计中会变化的因素。这个实现允许实现化角色有多态性变化。换言之，客户端可以持有抽象化类型的对象，而不在意对象的真实类型是“实现化 1”、“实现化 2”还是“实现化 3”，如图 1-3 所示。

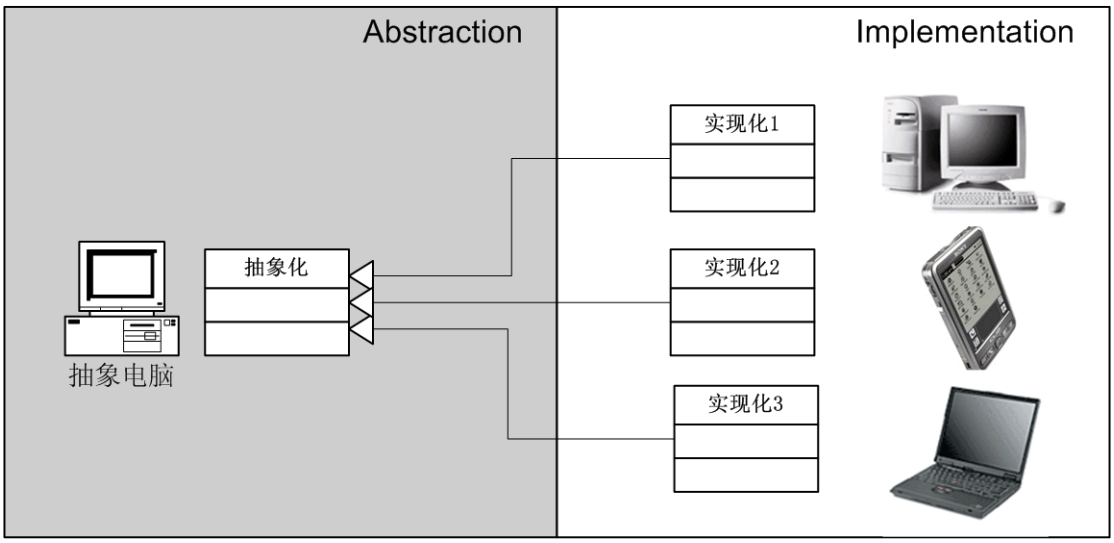


图 1-3 利用继承封装变化，实现复用

另外通过继承，一旦定义了电脑的特征，之后通过增加不同新方法和新属性就能定义台

式机和笔记本。显然，新的具体电脑对象都继承了 CPU、内存、硬盘、操作系统、应用软件等属性，以及开机、运行程序等行为，从而实现了有关代码的复用。

那么，在 Delphi 中，我们如何用面向对象编程的方法来实现这样一个简单的应用呢？下面我就给出具体的程序开发示例。

首先，我们遵循继承封装变化，实现复用的思路来设计相关的类。假设这里所有的电脑运行（Run）的步骤都是一样的，即：开机（PowerOn）、自检（SysCheck）、载入操作系统（LoadOS），这就是其抽象的业务逻辑，这是系统中不变的部分，可以作为基类。但是具体的实现行为可能不一样，比如笔记本电脑开机使用电池，而台式机则使用 220V 电源，因此我们需要设计具体的派生类来封装这些变化。

对于不易确定行为和多变的部分，我们最好在基类中将其设计成虚抽象方法，留给派生类去覆盖（override），也就是说把变化部分的具体实现延迟到派生类。示例程序中，PowerOn 就是这样一个变化部分

最后我们通过 UML 建模，设计的系统结构如图 1-4 所示。这一步我是在 Delphi 建模工具 ModelMaker 中完成的，ModelMaker 可以把我们设计的类图自动转化成代码框架，提高我们的编程效率。

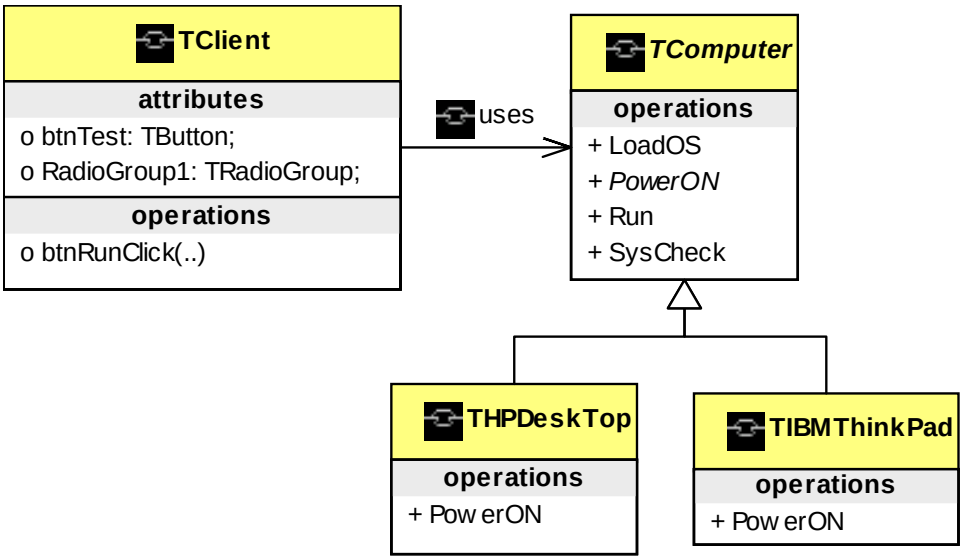


图 1-4 在电脑的继承结构中，基类封装了相对稳定的逻辑，派生类则封装了具体变化的因素

例程序 1-1 是 computerClass 单元的实现源代码。其实现代码如所示。从中我们看到抽象基类 TComputer 封装了相对稳定的逻辑，支持代码的复用。

例程序 1-1 computerClass 单元的实现源代码

```
unit computerClass;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
```

```
  TComputer = class (TObject)
  public
    procedure LoadOS;
    procedure PowerON; virtual; abstract;
    procedure Run;
    procedure SysCheck;
  end;

  THPDeskTop = class (TComputer)
  public
    procedure PowerON; override;
  end;

  TIBMThinkPad = class (TComputer)
  public
    procedure PowerON; override;
  end;

implementation

{抽象基类 TComputer 封装了相对稳定的逻辑，支持代码的复用}
procedure TComputer.LoadOS;
begin
  ShowMessage('载入 Windows 操作系统');
end;

procedure TComputer.Run;
begin
  PowerON;
  SysCheck;
  LoadOS;
end;

procedure TComputer.SysCheck;
begin
  ShowMessage('检测硬件系统');
end;

{具体派生类 THPDeskTop 和 TIBMThinkPad 封装了易变化的部分，这部分行为可以动态绑定，不影响客户端，便于维护}

{THPDeskTop}
procedure THPDeskTop.PowerON;
begin
```

```
  ShowMessage('开启 HP 台式电脑，电压 220V，工作正常');
end;

{TIBMThinkPad }
procedure TIBMThinkPad.PowerON;
begin
  ShowMessage('开启 IBM ThinkPad 笔记本电脑，剩余电池 80%');
end;


end.
```

以其 Run 方法为例，该方法封装了电脑运行的逻辑。尽管其中的 PowerON 方法在基类 TComputer 还不知道是如何实现的。但这也不妨碍基类的稳定与复用。

```
procedure TComputer.Run;
begin
  PowerON;
  SysCheck;
  LoadOS;
end;
```

请注意，具体派生类 THPDeskTop 和 TIBMThinkPad 封装了变化的部分，这部分行为可以动态绑定，不影响客户端，便于维护。

为什么这样说？这是因为在客户端声明的电脑对象 Computer 是 TComputer 类型，无论具体的电脑是 HPDeskTop 还是 IBMThinkPad，客户端调用的 Computer 的 Run 方法都是一样的。派生类 THPDeskTop 和 TIBMThinkPad 都可以通过向上转型去实现从 TComputer 继承的所有方法，而客户端无需去具体控制他们，更无需了解他们是如何实现不同的变化的。即使以后新增了一种具体的电脑类型，例如：掌上电脑 TPalmTop，只需在 computerClass 单元的增加实现 TPalmTop 的代码，而客户端的改动仅仅是增加一条代码：

```
Computer:= TPalmTop.Create;
```

客户端调用的 Computer 的 Run 方法还是一样的，这就是多态性的应用。客户端使用电脑的示意代码如示例程序 1-2 所示。

示例程序 1-2 UserForm 单元的源代码

```
unit UserForm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls,Bridge, ExtCtrls,computerClass;
```

```
type
  TClient = class (TForm)
    btnRun: TButton;
    RadioGroup1: TRadioGroup;
    procedure btnRunClick(Sender: TObject);
  end;

var
  Client: TClient;

implementation

{$R *.dfm}

{TClient }
procedure TClient.btnRunClick(Sender: TObject);
var
  Computer: TComputer;
begin
  //客户端（TClient）使用电脑（TComputer）的示意代码
  try
    case RadioGroup1.ItemIndex of
      0: Computer:=TIBMThinkPad.Create;
      1: Computer:=THPDeskTop.Create;
    end;
    Computer.Run;
  finally
    Computer.Free;
  end;
end;


end.
```

1.3.3 借助模式封装多个变化

继承是类型的复用，实现较为容易。往往初学面向对象的程序员十分热衷使用。但其存在不少缺点，例如：继承将基类的实现细节暴露给派生类可能破坏封装；基类的实现发生改变，那么派生类的实现也不得不发生改变等。特别是面对一些复杂问题，这种利用继承实现的简单的面向对象编程显然无法胜任。

在前面的示例程序中，一个继承关系封装了一个变化因素（即 PowerOn）。这种做法是正确的，它符合**单一职责法则（SRP）**，即一个继承关系不应当同时处理两个变化因素。换

言之，这种简单实现不能够处理抽象化与实现化都面临变化的情况。

当抽象化与实现化都面临变化时，例如在前面的示例程序中，不仅仅存在具体电脑硬件系统上的变化（例如 PowerOn），还存在抽象电脑软件系统上的变化（例如 LoadOS）时，为了封装变化和支持复用，我们所面临的问题就无法通过单一的继承来解决。

当然，如果两个变化因素是彼此孤立的，可以在不影响另一者的情况下独立演化。假设电脑的软件系统和硬件系统彼此独立，即任何硬件系统都支持任何软件系统。则可以如图 1-5 所示，由抽象电脑（软件系统）和具体电脑（硬件系统）两个类层次结构分别封装了各自的变化因素，例如软件系统的不同操作系统和硬件系统的不同 CPU。由于每一个变化因素都是可以通过静态关系表达的，因此分别使用继承关系实现。

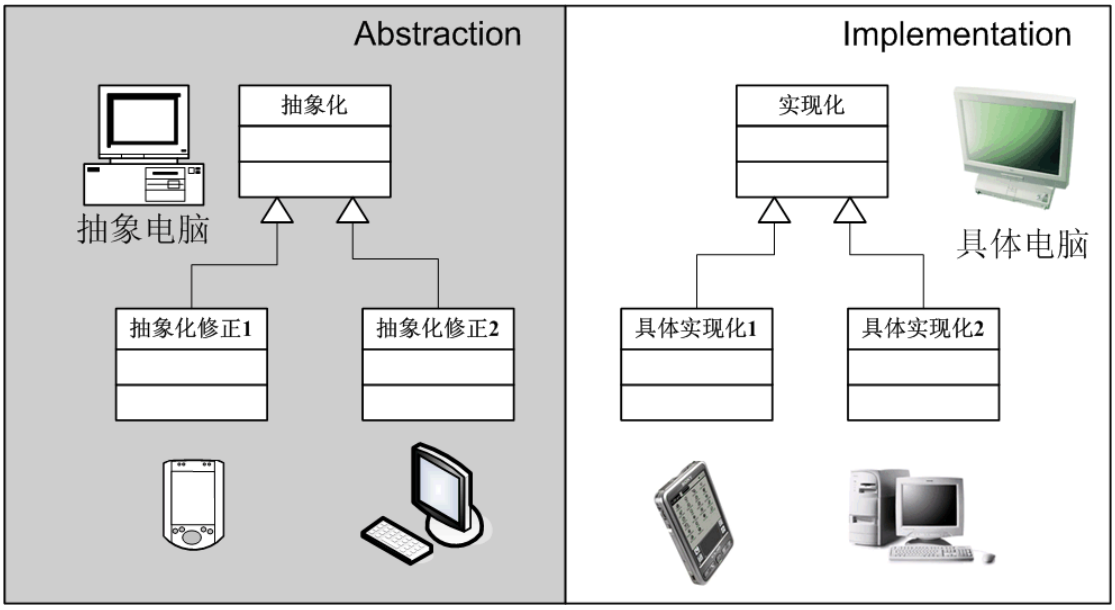


图 1-5 出现两个不同的独立演化方向

如果一个电脑所涉及到的软件系统和硬件系统这两个变化因素是相互依赖的怎么办？也就是说，存在抽象化与实现化之间的变化怎么办呢?虽然可能会有很多种设计，但能够真正体现封装变化、实现复用的好设计往往只有一个，这就是基于模式的设计。

正确的设计方案应当是使用两个独立的类层次结构封装两个独立的变化因素，并在它们之间使用合成关系，以达到功能复合的目的。有模式编程经验的程序员就自然地会想到桥接（Bridge）模式。

如图 1-6 所示，我们可以把电脑分别分为依赖于软件系统变化的抽象电脑和依赖于硬件系统变化的电脑。在抽象化与实现化的变化分别得到封装之后，再使用合成关系关联抽象化角色与实现化角色，从而提供满足客户端需求的一个完整的电脑。我们从图 1-6 中看到桥接模式连接两个不同的类型，支持两个不同方向上的演化，是沟通两种不同方向上变化的桥梁。

桥接模式用于将抽象化与实现化脱耦，使得二者可以独立地变化。由于桥接模式把抽象部分和它的实现部分分离，而且实现类的接口又与其具体实现分离，这样我们就可以改变或替换一个程序的实现而不用改变客户端的代码，从而达到软件易维护，可复用的效果。

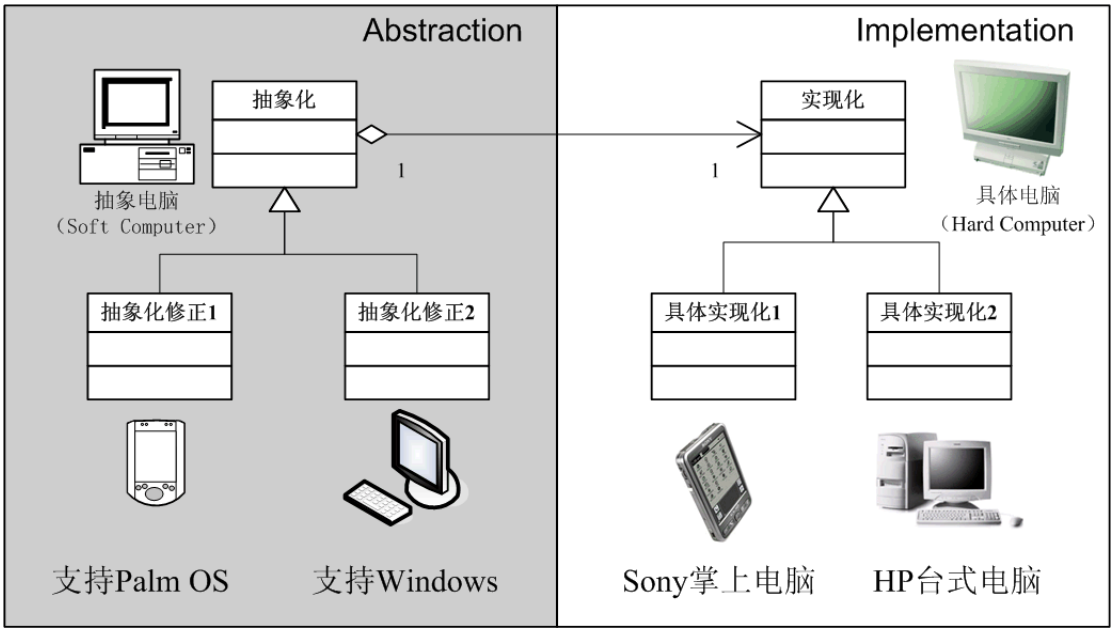


图 1-6 桥接模式连接两个不同的类型，支持两个不同方向上的演化

示例程序基于桥接模式模式的具体设计如图 1-7 所示。我们可以把依赖于软件系统变化的抽象电脑和依赖于硬件系统变化的电脑分别声明为基类 TSoftComputer 和 THardComputer，其派生类分别封装了对应的变化因素，而且还可以继续扩充。TSoftComputer 和 THardComputer 之间是合成关系。

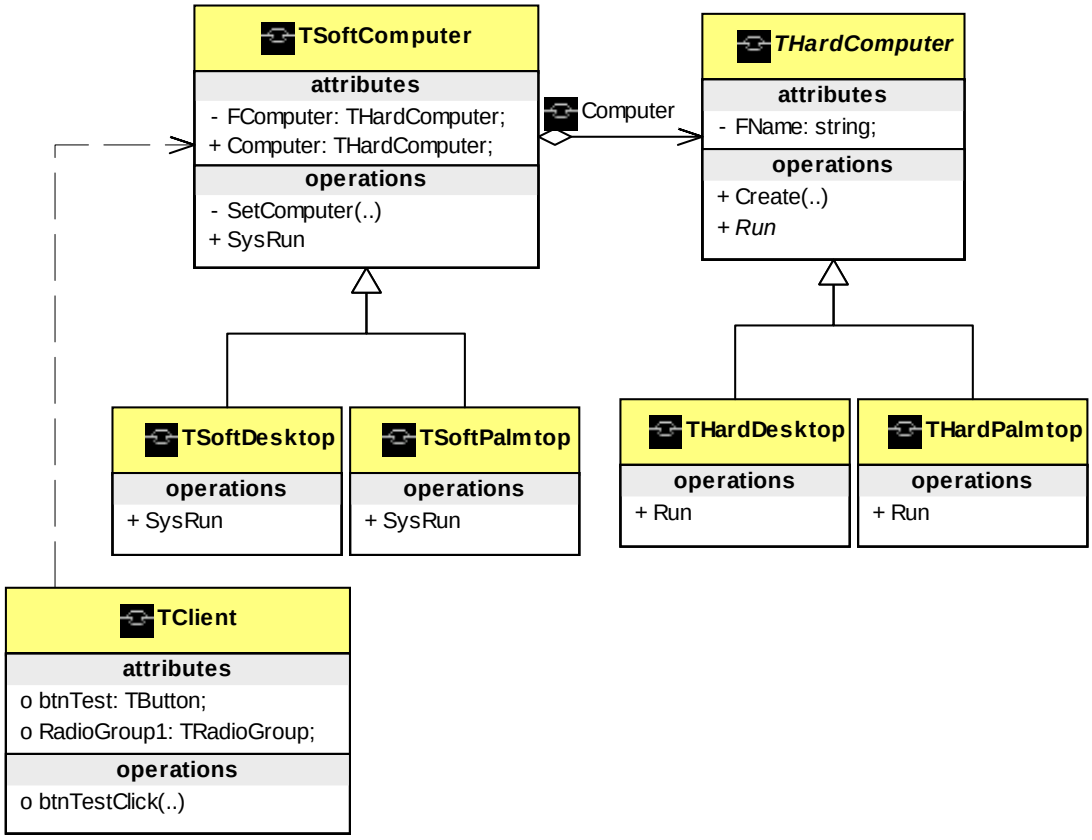


图 1-7 电脑示例程序基于桥接模式模式的设计

示例程序 1-3 是桥接模式的应用实现代码。抽象电脑 TSoftComputer 通过 SetComputer 方法使 FComputer 获得具体电脑对象（THardComputer 的实例）的引用，并在 SysRun 虚方法中调用硬件系统的 Run 方法。其派生类在覆盖并实现自己的 SysRun 方法时继承了基类的 SysRun 方法，这就意味着他们的 SysRun 方法除了得到了各自的硬件系统的运行行为还有与之对应的软件系统运行行为。通俗地讲，通过桥接模式，我们组合了软件系统和硬件系统的两种特征因素（也是主要的变化因素），从而得到了满足用户需求的特定的一个具体电脑系统。这种组合可以在运行期动态进行，如示例程序 1-4 所示。

示例程序 1-3 Bridge 单元的源代码

```
unit Bridge;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  THardComputer = class (TObject)
  private
    FName: string;
  public
    constructor Create(name:string);
    procedure Run; virtual; abstract;
  end;

  THardDesktop = class (THardComputer)
  public
    procedure Run; override;
  end;

  THardPalmtop = class (THardComputer)
  public
    procedure Run; override;
  end;

  TSoftComputer = class (TObject)
  private
    FComputer: THardComputer;
    procedure SetComputer(Value: THardComputer);
  public
    procedure SysRun; virtual;
    property Computer: THardComputer read FComputer write SetComputer;
  end;
```

```
  TSoftPalmtop = class (TSoftComputer)
  public
    procedure SysRun; override;
  end;

  TSoftDesktop = class (TSoftComputer)
  public
    procedure SysRun; override;
  end;

implementation


{ THardComputer }
constructor THardComputer.Create(name:string);
begin
  FName:=name;
end;

{ THardDesktop }
procedure THardDesktop.Run;
begin
  ShowMessage(FName+'开机运行，加载 220V 电源...');
  ShowMessage('台式机：'+FName+'硬件工作正常！');
end;

{ THardPalmtop}
procedure THardPalmtop.Run;
begin
  ShowMessage(FName+'开机运行,接通锂电池...');
  ShowMessage('掌上电脑：'+FName+'硬件工作正常！');
end;

{TSoftComputer }
procedure TSoftComputer.SetComputer(Value: THardComputer);
begin
  FComputer:=Value;
end;

procedure TSoftComputer.SysRun;
begin
  FComputer.Run;
end;
```

```
{TSoftPalmtop }
procedure TSoftPalmtop.SysRun;
begin
  inherited;
  //自己的修正代码
  ShowMessage('加载 Palm OS 操作系统...');
  ShowMessage('掌上机软件系统运行正常!');
end;

{TSoftDesktop }
procedure TSoftDesktop.SysRun;
begin
  inherited;
  //自己的修正代码
  ShowMessage('加载 Windows 操作系统...');
  ShowMessage('台式机软件系统运行正常!');
end;

end.
```

示例程序 1-4 是客户端应用程序。我们看到该程序代码简洁，目的明确。客户端无需知道作为抽象化的电脑软件系统和作为实现化的电脑硬件系统是如何变化与运行的，客户端只需为作为接口的抽象电脑对象 AbstractComputer 指定一个具体电脑的类型即可，其软硬件系统将自动匹配。

示例程序 1-4 ClientForm 的源代码

```
unit ClientForm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls,Bridge, ExtCtrls;

type
  TClient = class (TForm)
    btnTest: TButton;
    RadioGroup1: TRadioGroup;
    edtName: TLabeledEdit;
    procedure btnTestClick(Sender: TObject);
  end;

var
```

```
  Client: TClient;

implementation

{$R *.dfm}

{TClient }
procedure TClient.btnTestClick(Sender: TObject);
var
  AbstractComputer:TSoftComputer;
  Computer:THardComputer;
begin
  try
    if  RadioGroup1.ItemIndex=0 then
    // 掌上机
    begin
      AbstractComputer:=TSoftPalmtop.Create;
      AbstractComputer.Computer:=THardPalmtop.Create(edtName.Text);
    end
    else
    //台式机
    begin
      AbstractComputer:=TSoftDesktop.Create;
      AbstractComputer.Computer:=THardDesktop.Create(edtName.Text);
    end;
    AbstractComputer.SysRun;
  finally
    AbstractComputer.Computer.Free;
    AbstractComputer.Free;
  end;
end;

end.
```

通过程序我们看到，软硬件系统的变化组合可以是在运行期动态进行的。假设有某种台式机也可以模拟掌上电脑运行，我们不妨稍加改动，写下以下代码即可：

```
    // 台式电脑模拟 Palm 掌上机
    begin
      AbstractComputer:=TSoftPalmtop.Create;
      AbstractComputer.Computer:= THardDesktop.Create(edtName.Text);
    end
```

从更高的层次讲，一个好的设计中继承结构通常不多于两层，这也是单一职责法则（SRP）的要求。如果出两个以上的变化因素，就需要找出哪一个因素是静态的，可以使用继承关系；哪一个是动态的，必须使用合成关系。而桥接模式作为高手开发经验的总结为我们提供了一个快速的解决方案。

1.3.4 模式帮助我们解决问题

面向对象编程比较困难，而设计可复用的面向对象软件就更加困难。你必须找到相关的对象，以适当的粒度将它们归类，再定义类的接口和继承层次，建立对象之间的基本关系。你设计时既要针对手头问题的特殊性，又要对将来的问题和需求留有足够的变通性。

尽管大家都希望避免重复设计或尽可能少做重复劳动，但有经验的面向对象开发者会告诉你，要一下子就得到复用性和灵活性好的设计，即使不是不可能的至少也是非常困难的。一个设计在最终完成之前常要被复用好几次，而且每一次都有所修改。

有经验的面向对象开发者的确能做出良好的设计，而新手则面对众多选择无从下手，总是求助于以前使用过的非面向对象技术。新手需要花费较长时间领会良好的面向对象设计是怎么回事。有经验的开发者显然知道一些新手所不知道的东西，其中的奥妙就在于他们掌握了模式编程或与其相似的经验。

例如封装变化是面向对象设计中的难点，它也是很多模式的主题。通过深入研究我们还能进一步发现，当一个程序的某个方面的特征经常发生改变时，这些模式就定义一个封装这个方面的对象。这样当该程序的其他部分依赖于这个方面时，它们都可以与此对象协作。这些模式通常定义一个抽象类来描述这些封装变化的对象，并且通常该模式依据这个对象来命名。例如：

- 用 Strategy 模式的策略对象封装一个算法。
- 用 State 模式的状态对象封装一个与状态相关的行为。
- 用 Mediator 模式的中介者对象封装对象间的协议。
- 用 Iterateor 模式的迭代子对象封装访问和遍历一个聚集对象中的各个构件的方法。

这些模式描述了程序中很可能会改变的方面，并给出了封装变化的宝贵经验。通过应用这些模式，我们可以轻松解决封装的问题，而不必进行重复的探索。

模式使人们可以更加简单方便地复用成功的设计和架构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。设计模式帮助你做出有利于系统复用的选择，避免设计损害了系统复用性。简而言之，模式编程可以帮助我们更快更好地完成系统。

通过本节这个由面向对象编程到模式编程的演化实例，我们可以进一步认识到模式编程不是空洞的理论和时髦的词汇，模式编程是面向对象编程的深入和提高，模式是面向对象编程高手经验的总结，是学习面向对象编程精华之所在。

第8章　原型模式（Prototype）

8.1　模式解说

原型模式通过给出一个原型对象来指明所要创建对象的类型，然后克隆该原型对象以便创建出更多同类型的新对象。

例如：在 Delphi 的 IDE 中，我们为设计窗体拖放了一个按钮对象。为了快速创建更多的同样字体和尺寸的按钮对象，我们可以复制该按钮（使用菜单 Copy 菜单或快捷键 Ctrl+C），并在设计窗体多次粘贴（使用菜单 Paste 菜单或快捷键 Ctrl+V），如图 4-1 所示。

设计窗体中的按钮对象是用于我们应用程序的，而 IDE 中提供的按钮对象创建方法（复制和粘贴）则是属于 Delphi 架构的。我们通过复制创建一个按钮对象时，不需要知道 Delphi 是如何实现的。

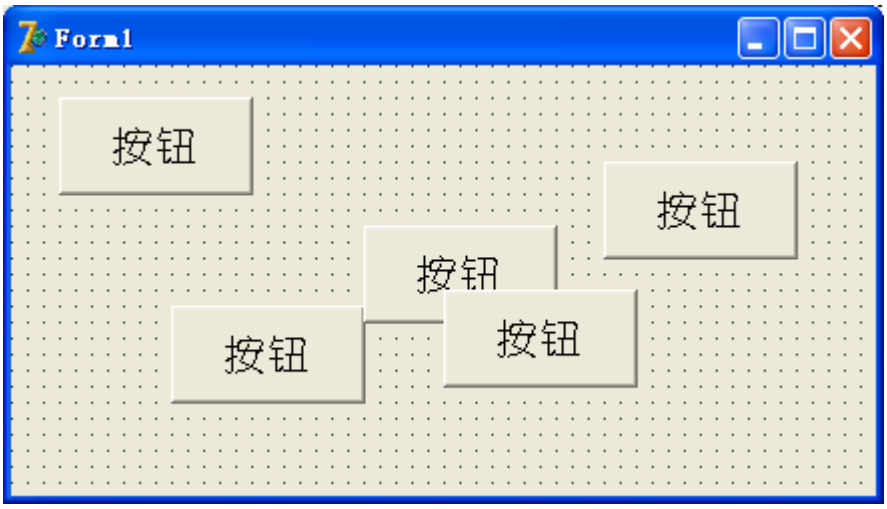


图 8-1 在 Delphi 设计窗体上复制按钮对象

所要说明的是，虽然我们使用的是类似文字处理中的复制和粘贴，但复制的决不是一个按钮对象的外观（字体和尺寸等），而是整个按钮对象，包括它的属性和方法。所以，更严格讲，我们是克隆了这个对象，即得到一个和源对象一样的新对象。我们称这种被克隆的对象（比如按钮）为原型。只要系统支持克隆功能，我们就可以任意克隆对象。

由此可见，原型模式适用于系统应该与其对象的创建、组合及显示时无关的情况，包括：

- 当要实例化的类是在运行时刻指定时，例如，通过动态载入。
- 当类实例只是少数不同组合状态其中之一时，这时比较好的方式在适当的状态下使用一些组合的原型并复制他们，而不是人工的继承这些类。
- 避免建立工厂类等级结构平行产出类等级结构时。

假设一个系统的产品类是动态加载的，而且产品类具有一定的等极结构。这个时候如果采取工厂模式的话，工厂类就不得不具有一个相应的等级。而产品类的等级结构一旦变化，工厂类的等级结构就不得不有一个相应的变化。这对于产品结构可能会有经常性变化的系统来说，采用工厂模式就有不方便之处。

这时如果采取原型模式，给每一个产品类配备一个克隆方法（大多数的时候只需给产品

类等级结构的基类配备一个克隆方法），便可以避免使用工厂模式所带来的具有固定等级结构的工厂类。

这样，一个使用了原型模式的系统与它的产品对象是如何创建出来的，以及这些产品对象之间的结构是怎样的，还有这个结构会不会发生变化，都是没有关系的。

8.2 结构和用法

8.2.1 模式结构

原型模式的结构如图 8-2 所示。它涉及到三个参与者：

- 抽象原型（TPrototype）——声明一个接口以便克隆其本身。
- 具体原型（TConcretePrototype）——实现克隆的操作以克隆本身。
- 客户（TClient）——请求原型克隆其本身以构建新的对象。

他们之间的合作关系为客户请求原型克隆复制其本身以构建新的对象。

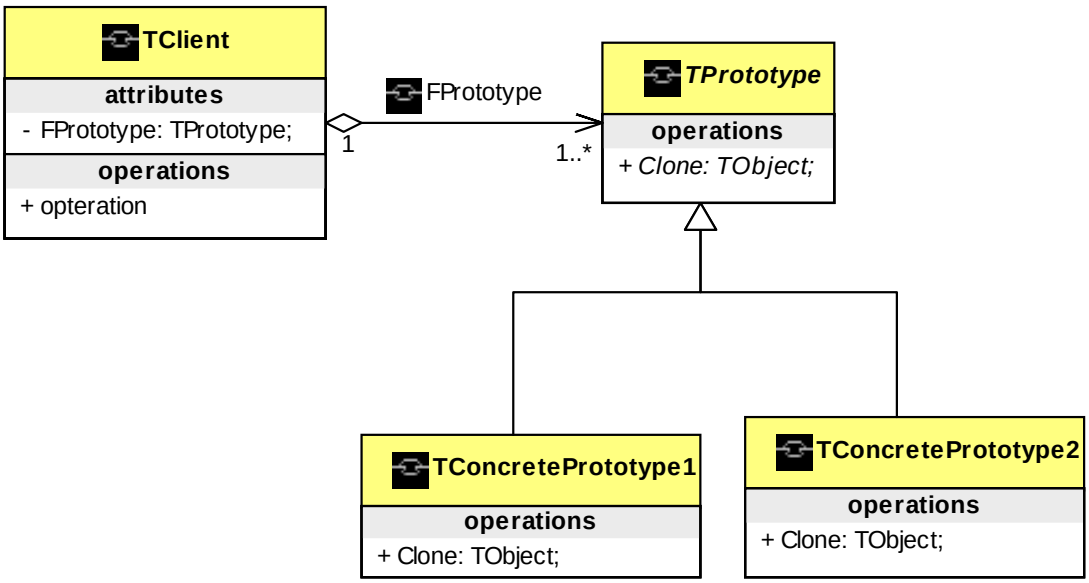


图 8-2 原型模式的结构图

8.2.2 代码模板

原型模式的实现代码模板如示例程序 8-1 所示。

示例程序 8-1 原型模式的实现代码模板

```
unit Prototype;

interface

uses
```

```
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type

  TPrototype = class (TObject)
  public
    function Clone: TObject; virtual; abstract;
  end;

  TConcretePrototype1 = class (TPrototype)
  public
    function Clone: TObject; override;
  end;

  TConcretePrototype2 = class (TPrototype)
  public
    function Clone: TObject; override;
  end;

  TClient = class (TObject)
  private
    FPrototype: TPrototype;
  public
    procedure opteration;
  end;

implementation

{TClient }
procedure TClient.opteration(NewObj:TPrototype);
var
  NewObj:TPrototype;
begin
  {克隆一个对象}
  NewObj:=FPrototype.Clone;
end;

{TConcretePrototype1
function TConcretePrototype1.Clone: TObject;
begin
{克隆的实现代码}
end;
```

```
{TConcretePrototype2}
function TConcretePrototype2.Clone: TObject;
begin
{克隆的实现代码}
end;

end.
```

前面我介绍的原型模式通常用于原型对象数目较少且比较固定的情况，这种情况下原型对象的引用由客户对象自己保存。但是，如果要创建的原型对象数目不是很固定，则可以采用下面介绍的注册形式的原型模式。在这种情况下，客户对象不用保存对原型对象的引用，而是另有原型管理器负责。

我们把负责注册原型对象的参与者称为原型管理器（prototype manager）。原型管理者储存原型并依据一定的键值（或索引值）传回相对应的原型，它包含了原型对象的添加、删除等操作，并使每个原型对象相对应一个键值（或索引值）。客户端可以在运行期改变或者浏览这个注册项。这样一来，客户端在系统中管理及扩充原型对象数量都无须撰写更多的程序代码。Delphi 的 TObjectList 类可以帮我们实现原型对象的注册和管理。

图 8-3 是注册形式的原型模式结构图。它比图 8-2 多了一个原型管理器（TPrototypeManager）。原型管理器负责创建具体原型类的对象，并记录每一个被创建的对象。

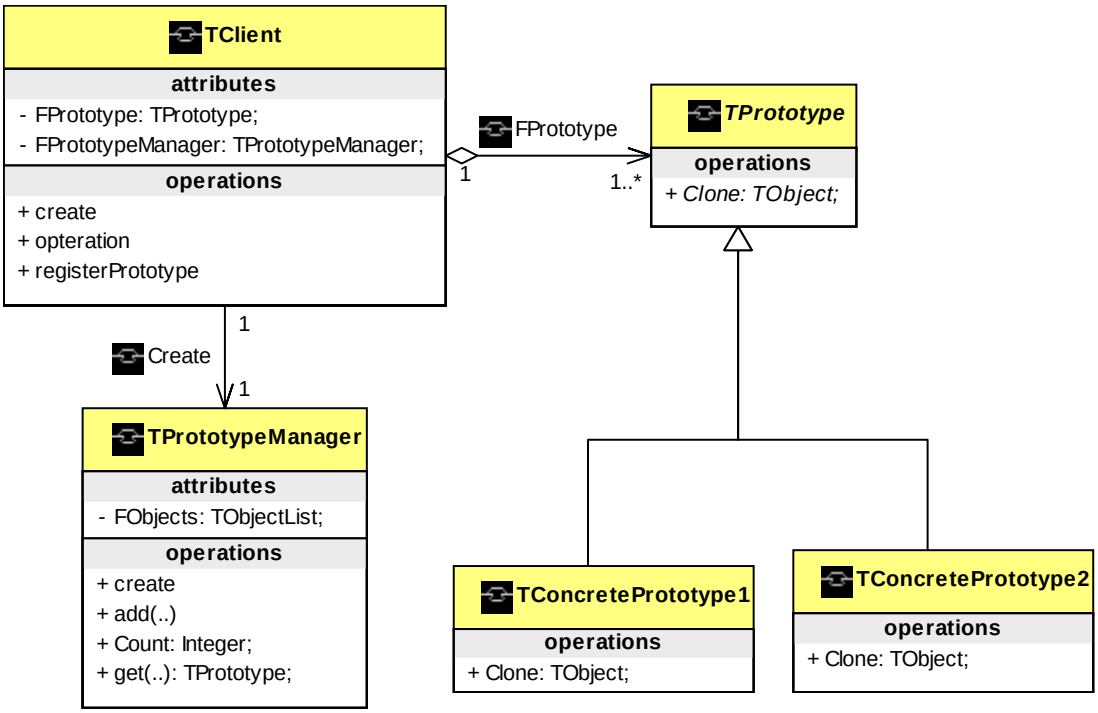


图 8-3 注册形式的原型模式的结构图

示例程序 8-2 给出了一个示意性实现的源代码。首先，抽象原型 TPrototype 声明了一个 Clone 方法，然后由具体原型 TConcretePrototype1 和 TConcretePrototype2 分别实现该 Clone 方法。管理器 TPrototypeManager 为注册原型对象提供必要的方法，包括：新增、获取、计数等。它实际上是通过 Delphi 的 TObjectList 自动实现这些功能的。

系统中的客户对象通常先创建一个新的原型对象，然后克隆一份，注册并保存在原型管理器中。在注册形式的原型模式中，当客户对象克隆一个原型对象之前，客户对象先查看管理对象中是否已经存在有满足要求的原型对象。如果有就直接从管理对象中取得该对象的引用；如果没有，则克隆并注册该原型对象。

示例程序 8-2 注册形式的原型模式实现代码模板

```
unit Prototype2;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs,Contnrs;

type
  TPrototype = class (TObject)
  public
    function Clone: TObject; virtual; abstract;
  end;

  TConcretePrototype1 = class (TPrototype)
  public
    function Clone: TObject; override;
  end;

  TConcretePrototype2 = class (TPrototype)
  public
    function Clone: TObject; override;
  end;

  TPrototypeManager = class (TObject)
  private
    FObjects: TObjectList;
  public
    constructor create; override;
    procedure add(AObject: TPrototype);
    function Count: Integer;
    function get(Index: Integer): TPrototype;
  end;

  TClient = class (TObject)
  private
    FPrototype: TPrototype;
    FPrototypeManager: TPrototypeManager;
```

```
  public
    constructor create; override;
    procedure opteration;
    procedure registerPrototype;
  end;

implementation

{TClient }
procedure TClient.opteration;
begin
//具体实现代码
end;

constructor TClient.create;
begin
  FPrototypeManager:=TPrototypeManager.Create;
end;

procedure TClient.registerPrototype;
var
  ClonedObj:TPrototype;
  i:integer;
begin
   //示意性代码
    FPrototype:=TConcretePrototype1.Create;
    ClonedObj:=TPrototype(FPrototype.Clone);
    FPrototypeManager.add(ClonedObj);
end;

{TConcretePrototype1 }
function TConcretePrototype1.Clone: TObject;
begin
//具体实现代码
end;

{TConcretePrototype2}
function TConcretePrototype2.Clone: TObject;
begin
//具体实现代码
end;

{TPrototypeManager}
constructor TPrototypeManager.create;
```

```
begin
  FObjects.Create;
end;

procedure TPrototypeManager.add(AObject: TPrototype);
begin
  FObjects.Add(AObject);
end;

function TPrototypeManager.Count: Integer;
begin
  result:=FObjects.Count;
end;

function TPrototypeManager.get(Index: Integer): TPrototype;
begin
  result:=TPrototype(FObjects.Items[Index]);
end;

end.
```

原型模式有许多与抽象工厂模式和建造者模式相同的效果，包括客户端不知道具体产品类，而只知道抽象产品类，客户端不需要知道很多的具体产品名称。如果有新的产品类加入，客户端不需要进行改动就可直接使用。

8.3 范例与实践

8.3.1 Delphi 对象的克隆

原型模式通过克隆原型对象来创建新对象，因此了解和掌握 Delphi 中对象的克隆是使用原型模式的关键。

在 Delphi 创建一个对象实际上就是把一个类进行实例化。例如要从 TMan 类创建一个名为 Tom 的对象，可以这样创建：

```
var Tom:TMan;
......
Tom:=TMan.Create;
```

以上语句完成了以下工作：

- 声明 TMan 类型的变量 Tom；
- 为 TMan 类创建一个实例；
- 将变量 Tom 指向创建的实例。

我们从中可以发现，对象变量和对象并不是一回事。对象是 TMan 类创建的一个实例，对象变量是该对象的引用。为了简单，在称呼上我们通常并不严格区分。但在使用时，务必分清对象引用和实际对象。

有时在使用对象时无需使用对象变量来区分某一对象，例如：

```
Factory.MakeTool(TMan.Create);
```

这里无需区分 TMan 的实例是 Tom 还是 Jack。

但我们使用以下例子时，表示 Tom 和 Jack 分别引用了不同的 TMan 的实例，此时他们是两个对象。

```
var Tom,Jack:TMan;
......
Tom:=TMan.Create;
Jack:=TMan.Create;
```

但是如果接着使用以下语句：

```
Tom:=Jack;
```

此时 Tom 变量就不再引用 Tom 对象，而是引用 Jack 对象，这就好像 Tom 变成了 Jack 的另一个名字。当你找 Tom 时，找到的是 Jack。所以这种方法只能复制对象的引用而不能克隆整个对象。

由此我们了解到，对象是类的动态实例，对象总是被动态分配到堆上。因此一个对象引用就如同一个句柄或一个指针。但你分配一个对象引用给一个变量时，Delphi 仅复制引用，而不是整个对象。在 Delphi 中使用一个对象的唯一方法就是使用对象引用。一个对象引用通常以一个变量的形式存在，但是也有函数或者属性返回值的形式。

Delphi 中不像有的语言那样提供了对象克隆的功能（比如：Java 有 Object.clone 方法），所以在 Delphi 中实现对象克隆的功能需要自己编写代码。

好在 VCL 的体系结构中，TPersistent 类系下的对象可以通过覆盖 Assign 方法，实现克隆行为。TPersistent 的 Assign 方法较常用于两个对象属性的复制。在 Assign 方法中可以完成对象属性、方法和事件的逐个复制。

Assign 方法在 TPersistent 类中声明为虚方法，以便允许每个派生类定义自己的复制对象方法。如果派生类没有重写 Assign 方法，则 TPersistent 的 Assign 方法会将复制动作交给源对象来进行：

```
procedure TPersistent.Assign(Source: TPersistent);
begin
  if Source <> nil then Source.AssignTo(Self)
else AssignError(nil);
end;
```

由此可见， Assign 方法实际上是调用 AssignedTo 方法来实现的，因此 TPersistent 的 Assign 方法很少被派生类所重载，但 AssignTo 却常被派生类根据需要重载。

如果由 AssignedTo 方法来实现复制，那么必须保证源对象的类已经重写了 AssignedTo 方法，否则将抛出一个 AssignError 异常：

```
procedure TPersistent.AssignError(Source: TPersistent);
var
  SourceName: string;
begin
  if Source <> nil then
    SourceName := Source.ClassName else
    SourceName := 'nil';
  raise EConvertError.CreateResFmt(@SAssignError, [SourceName, ClassName]);
end;
```

那么在程序中是如何使用 Assign 方法实现对象克隆的呢？如果要让对象 b 克隆对象 a，则可以考虑以下赋值操作：

```
......
var
{
  假设这里的 TMyObject 是 TPersistent 的派生类，
  并实现了 Assign 方法。比如：TFont 。
}
  a,b:TMyObject;
begin
  a:= TMyObject.create;
{ 关于对象 a 的代码}
  ......
{ 开始克隆对象 a}
  b:= TMyObject.create;
  b.Assign(a);//对象 b 的属性和内容和对象 a 完全相同。
end;
```

由此可见，b:=a 意味着 b 是 a 的引用，即两者是同一对象。如果写成 b.Assign(a)，那么 b 是仍然一个独立的对象，其状态与 a 相同，也就可以看成是 b 克隆了 a。

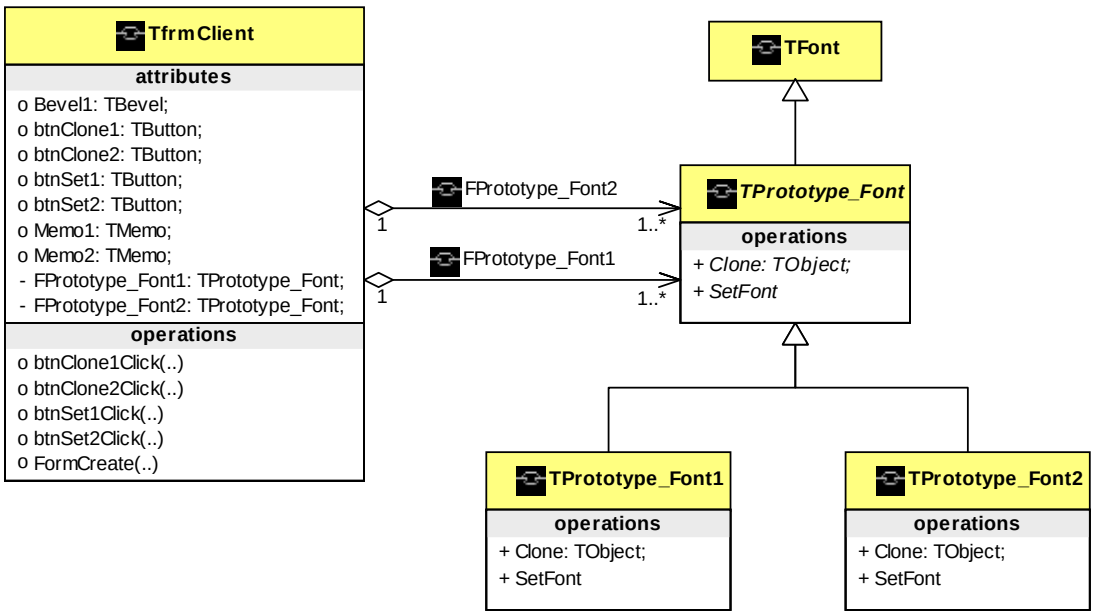
8.3.2 用原型模式克隆字体

在许多程序中我们需要允许用户自己定义他们喜欢的字体。几乎所有 Delphi 的用户界面控件（UI）都提供 TFont 属性，用于设置字体。另外 Delphi 还提供字体对话框方便可视化操作。通常在需要允许用户设置字体的地方，程序员可以写上这样一段代码：

```
if    FontDialog1.Execute then
   Memo1.Font.Assign(FontDialog1.Font);
```

如果在程序中有很多地方需要用户设置字体，这种代码就会出现在多处，并涉及到具体的用户界面控件，修改维护都很麻烦，而且用户也要费劲设置很多次字体。能不能让用户只设置一次字体，然后复制到他们需要的各处？也就是说能不能先创建一个与使用字体界面无关的字体对象原型，然后在需要的地方克隆？我们不妨就用原型模式来解决这个问题。

按照原型模式我设计的类图如图 8-2 所示。演示界面 TfrmClient 是客户，它使用抽象类 TPrototype_Font 提供的 SetFont 方法设置字体（原型对象）以及 Clone 函数克隆并返回克隆好的字体对象。但抽象类 TPrototype_Font 作为抽象原型仅仅是一个接口，真正实现 SetFont 方法和 Clone 函数的是具体原型 TPrototype_Font1 和 TPrototype_Font2。通过派生，TPrototype_Font1 类和 TPrototype_Font2 类可以提供不同的克隆产品和克隆实现方法。范例程序中我重点演示不同的克隆实现方法。



示例程序 8-3 范例程序的类图

示例程序 8-4 是原型字体程序的实现源码。在该程序中可以看到，我在 TPrototype_Font1 的 Clone 函数中调用了 Assign 方法来实现的字体克隆。TFont 实现了 TPersistent 的虚方法 Assign，作为 TFont 派生类的 TPrototype_Font1 理所当然继承 TFont 的 Assign 方法。

```
function TPrototype_Font1.Clone: TObject;
var
  Temp: TPrototype_Font1;
begin
  Temp:=TPrototype_Font1.Create;
  Temp.Assign(self);
  result:=Temp;
end;
```

如果在某些类中没有 Assign 方法（或没有实现 TPersistent 的 Assign 方法），我们就不得不自己撰写 Clone 函数了。TPrototype_Font2 中的 Clone 函数就是自己编程实现克隆的示例。

所谓克隆对象，实际上关键在复制该对象的状态，即对象的属性值（数据成员变量的值）。因为有同一个类创建的对象之间的差异就在于这些属性值的不同。

```
function TPrototype_Font2.Clone: TObject;
var
  Temp: TPrototype_Font2;
begin
  Temp:=TPrototype_Font2.Create;
  Temp.Name:=self.Name;
  Temp.Size:=self.Size;
  Temp.Color:=self.Color;
  result:=Temp;
end;
```

但是，有时候某些属性值并不是客户所需要的，所以在克隆时对对象的属性值进行取舍是允许的。TPrototype_Font2 中的 Clone 函数中，我只复制了影响字体外观的 3 个最主要的属性。这一取舍的后果是，克隆的新字体并不完全和原型字体一致，比如，不能反映出是否是斜体。这种克隆称为不完全克隆。

示例程序 8-4 原型字体程序的实现源码

```
unit PrototypeFont;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type

  TPrototype_Font = class (TFont)
  public
    function Clone: TObject; virtual; abstract;
    procedure SetFont; virtual; abstract;
  end;

  TPrototype_Font1 = class (TPrototype_Font)
  public
    function Clone: TObject; override;
    procedure SetFont; override;
  end;

  TPrototype_Font2 = class (TPrototype_Font)
```

```
  public
    function Clone: TObject; override;
    procedure SetFont; override;
  end;


implementation


{TPrototype_Font1 调用 Assign 方法实现的字体克隆。}
function TPrototype_Font1.Clone: TObject;
var
  Temp: TPrototype_Font1;
begin
  Temp:=TPrototype_Font1.Create;
  Temp.Assign(self);
  result:=Temp;
end;

procedure TPrototype_Font1.SetFont;
var
  FontDialog: TFontDialog;
begin
  FontDialog:= TFontDialog.Create(nil);
  try
    if  FontDialog.Execute then
      TFont(self).Assign(FontDialog.Font);
  finally
    FontDialog.Free;
  end;
end;

{TPrototype_Font2 通过自己编程实现的字体克隆。}
function TPrototype_Font2.Clone: TObject;
var
  Temp: TPrototype_Font2;
begin
  Temp:=TPrototype_Font2.Create;
  Temp.Name:=self.Name;
  Temp.Size:=self.Size;
  Temp.Color:=self.Color;
  result:=Temp;
end;
```

```
procedure TPrototype_Font2.SetFont;
var
  FontDialog: TFontDialog;
begin
  FontDialog:= TFontDialog.Create(nil);
  try
    if  FontDialog.Execute then
    begin
      self.Name:=FontDialog.Font.Name;
      self.Size:=FontDialog.Font.Size;
      self.Color:=FontDialog.Font.Color;
    end;
  finally
    FontDialog.Free;
  end;
end;

end.
```

在示例程序 8-5 所示的客户程序中，我设计了一个演示界面。通过“ 设置字体 ”按钮可以设置原型字体，然后点击“ <-克隆 ”按钮，可以将原型字体克隆到左边的文本编辑框中。如图 8-4 所示。

注意这里使用的多态和转型技术。TPrototype_Font 的 clone 函数把克隆的对象向下转型为 TObject 传出，示例程序 8-5 中再把返回的对象向上转型为 TPrototype_Font。因为 TFont 是 TPrototype_Font 的基类，下面的写法是安全的。

```
Memo1.Font:=TPrototype_Font(FPrototype_Font1.clone);
```

示例程序 8-5 客户程序源码

```
unit MainForm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls,PrototypeFont, ExtCtrls;

type
  TForm1 = class (TForm)
    Memo1: TMemo;
    btnSet1: TButton;
    btnClone1: TButton;
    btnClone2: TButton;
```

```
    Memo2: TMemo;
    btnSet2: TButton;
    Bevel1: TBevel;
    procedure FormCreate(Sender: TObject);
    procedure btnClone1Click(Sender: TObject);
    procedure btnSet1Click(Sender: TObject);
    procedure btnClone2Click(Sender: TObject);
    procedure btnSet2Click(Sender: TObject);
  private
    FPrototype_Font1: TPrototype_Font;
    FPrototype_Font2: TPrototype_Font;
  end;

var
  Form1: TForm1;

implementation



{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  FPrototype_Font1:=TPrototype_Font1.Create;
  FPrototype_Font2:=TPrototype_Font2.Create;
  Memo1.Lines.Add(
    '这里演示调用 Assign 方法实现的字体克隆。') ;
  Memo2.Lines.Add(
    '这里演示通过自己编程实现的字体克隆。') ;
end;

procedure TForm1.btnClone1Click(Sender: TObject);
begin
  Memo1.Font:=TPrototype_Font(FPrototype_Font1.clone);
end;

procedure TForm1.btnSet1Click(Sender: TObject);
begin
  FPrototype_Font1.SetFont;
end;

procedure TForm1.btnClone2Click(Sender: TObject);
var
```

```
  Prototype_Font2: TPrototype_Font2;
begin
  Memo2.Font:=TPrototype_Font(FPrototype_Font2.clone);
end;

procedure TForm1.btnSet2Click(Sender: TObject);
begin
  FPrototype_Font2.SetFont;
end;

end.
```

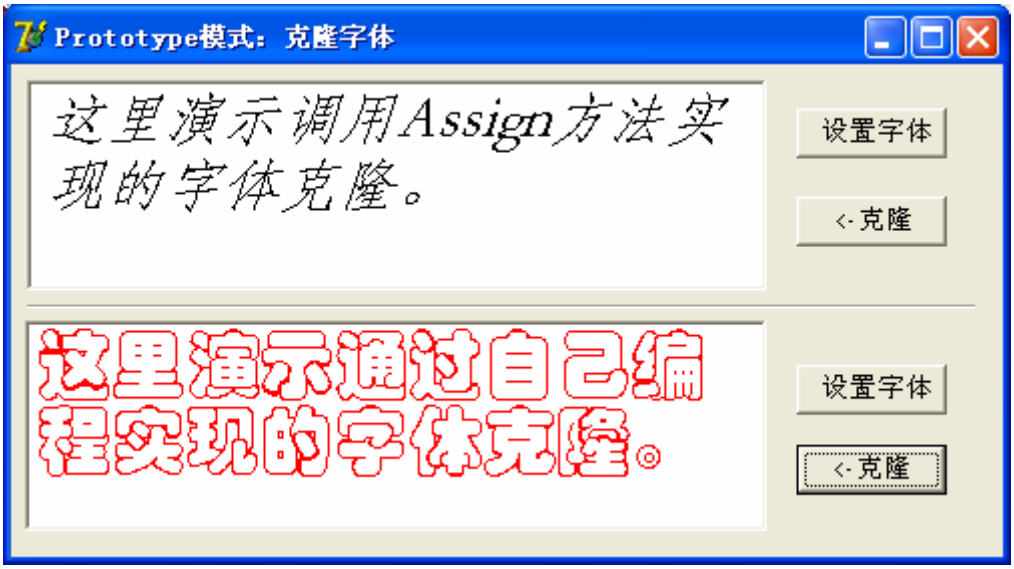


图 8-4 克隆字体的演示界面

8.3.3 Delphi 对象流化与原型模式

1. 使用流克隆对象实现原型模式

在 Delphi 中，流对象与流化存储技术不仅是 Delphi 可视化设计实现的核心，它也为实现原型模式提供了更为方便、简单和通用的对象克隆方法。尽管流化存储所涉及的存储媒介十分广泛，但在各对象的接口上得到了统一，使程序的存储操作变得十分方便、简单，从而使程序员能站在更高层面上进行对象存取的有关编程工作而无需考虑存储介质的具体差异。

在 VCL 中。从 TPersistent 类开始提供了一个接口，引入了对象的可赋值性和流化（assignment and streaming capabilities）等性质。TPersistent 类是抽象类，没有实例对象。但该类实现了对象公布（Published）属性的存取，即在该类及其派生类中声明为 Published 的属性、方法和事件等可在设计期时显示在 Object Inspector 窗中，能在 Object Inspector 中对对象的 Published 属性进行设计期的设计，并可将设置的值存到窗体或数据模块的 DFM 文件中。在程序运行期，对象将被初始化为设计期所设置的状态。例如，图 8-1 在 Delphi 窗

体上设计的按钮对象，在程序运行期该对象将被初始化为设计期所设置的状态。

在编程中为了实现流化，我们需要使用 TStream 的派生类。TStream 是所有流类的抽象基类，它继承自 TObject。它的派生类主要用于对文本、内存、数据库的 Blob 字段、数据压缩等进行操作。由于 TStream 与具体的存储无关，派生类却与存储媒介紧密相关，因此每个子类都必须实现与具体存储媒介相关的方法，如磁盘、内存等。通过流的写方法我们可以把对象转化成位模式保存在内存或文件中；同样，通过流的读方法我们可以把保存在内存或文件中的位模式重新转化成对象。通常把对象写到流里的过程称为串行化，而把对象从流中读出来的过程称为并行化。通过对象的一进一出，实际上就实现了对象的克隆。

需要注意的是，能够被流化的对象必须是 TPersistent 的派生类。一般 Delphi 的组件（TComponent 的派生类）都是支持流化的。用户要使自己定义的类能够支持流化，通常至少应该使其继承自 TPersistent。

> 参阅：需要进一步了解Delphi对象流化机制以及VCL相关的类，请参阅我著的《Delphi面向对象编程思想》（机械工业出版社2003年9月）。

下面我通过一个例子来演示说明如何使用流克隆对象实现原型模式。这个例子有点类似图 8-1 所示的通过 Delphi 的 IDE 复制和粘贴按钮对象。图 8-4 是演示程序的运行界面。我们点击“克隆对象”按钮，就会把一个文本框克隆到窗体上。

图 8-5 通过流技术克隆对象实现原型模式的演示程序

为此，我使用原型模式设计了图 8-6 的类结构。与前面原型模式示例程序不同的是，这里的 TMemoPrototype 类 Clone 方法利用了对象流化来完成对象的克隆。

```
function TMemoPrototype.Clone: TObject;
var
  tmp:TMemoPrototype;
  TmpStream:TStream;
begin
  WriteComponentResFile('MemoPrototype.dat',self);
```

```
  tmp:=TMemoPrototype(ReadComponentResFile('MemoPrototype.dat',nil));
  result:=tmp;
end;
```

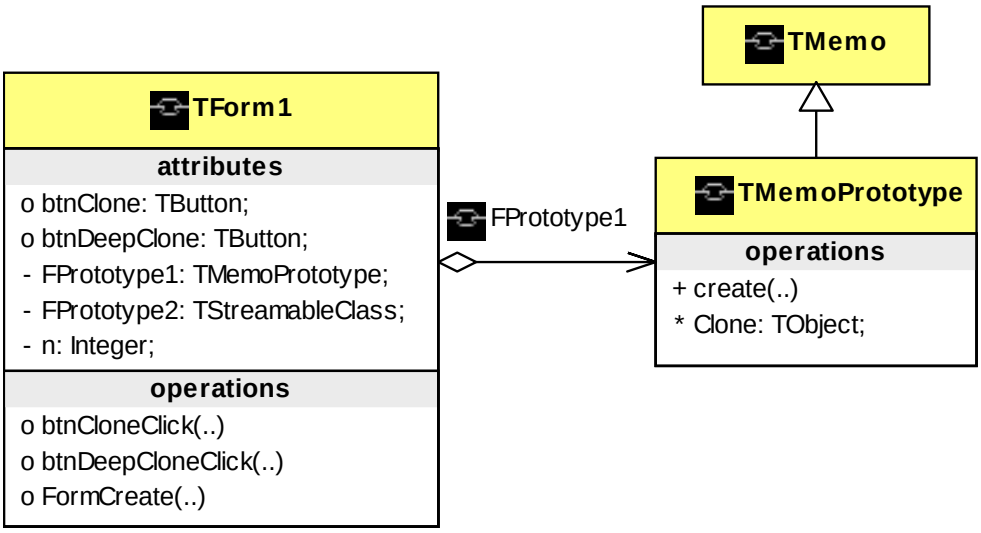


图 8-6 通过流技术克隆对象实现原型模式的类图设计

这里我们使用了 Delphi 的 Classes 单元中两个全局函数 WriteComponentResFile 和 ReadComponentResFile 来完成对象的串行化和并行化过程，前者将对象以位模式写到资源文件 MemoPrototype.dat 中，后者从资源文件 MemoPrototype.dat 中读出对象，每读出一个对象就好像克隆出了一个对象。这两个函数封装了文件流（TFileStream）的操作过程，使我们编程简单化，不过从他们的实现代码中（示例程序 8-6），我们还是可以看出这一流化过程的。

示例程序 8-6 WriteComponentResFile 和 ReadComponentResFile 的实现代码

```
procedure WriteComponentResFile(const FileName: string;
                        Instance: TComponent);
var
  Stream: TStream;
begin
  Stream := TFileStream.Create(FileName, fmCreate);
  try
    Stream.WriteComponentRes(Instance.ClassName, Instance);
  finally
    Stream.Free;
  end;
end;

function ReadComponentResFile(const FileName: string;
              Instance: TComponent): TComponent;
var
  Stream: TStream;
```

```
begin
  Stream := TFileStream.Create(FileName, fmOpenRead or fmShareDenyWrite);
  try
    Result := Stream.ReadComponentRes(Instance);
  finally
    Stream.Free;
  end;
end;
```

于是，在使用克隆对象的演示窗体中，我们通过点击“克隆对象”按钮，触发了以下按钮事件：

```
procedure TForm1.btnCloneClick(Sender: TObject);
var
  tmp:TMemoPrototype;
begin
  tmp:=TMemoPrototype(FPrototype1.clone);
  tmp.Name:='Memo'+IntToStr(n);
  tmp.Parent:=self;
  tmp.Lines.Add('克隆文本框之'+IntToStr(n));
  tmp.Left:=tmp.Left+30*n;
  tmp.Top:=tmp.Top+30*n;
  inc(n);
end;
```

通过克隆原型对象的方法，我们将方便地得到 TMemoPrototype 的实例。这些对象有着和原型对象一样的状态，比如：相同的外观和字体。（为了便于演示，我有意调整了他们的位置，否则窗体上的文本框会重叠在一起。）

由此可见，流化对象的强大之处在于无需复杂的处理就可以将对象和一个二进制流之间进行互相转化。这一功能可以巧妙地被我们用于对象克隆，特别是当某些对象不提供 Assign 方法实现时。

2. 对象的深克隆

前面我讲过，当克隆一个对象时，如果克隆的是该对象所有的状态，即被克隆对象的所有变量都含有与原始对象相同的值，称为完全克隆。否则，有选择地克隆原始对象的部分状态，只能称为不完全克隆。这是从对象克隆的广度上看问题。如果从对象克隆的深度上看，对象克隆还可以分成浅克隆和深克隆。所谓浅克隆仅仅克隆所考虑的对象，而不深入克隆它所引用的对象。如果在克隆当前对象时，同时也克隆了该对象所引用的对象，这就是所谓的深克隆了。在浅克隆中，被克隆的对象所引用的对象仍然是原始对象所引用的那个对象；而深克隆中，被克隆的对象所引用的对象已经是一个新的对象。这就是说，深克隆同时也克隆了引用对象本身，而不是它的一个引用。由于深克隆间接复制了引用的对象，如果出现循环引用，可能会出现意想不到的问题。所以当一个类引用了不支持串行化的间接对象，或者引用含有循环结构的时候，则不能考虑使用深克隆。

浅克隆和深克隆问题也称为浅复制和深复制（shallow copy versus deep copy）的问题，

该问题的焦点在于是否复制实例变量，亦或只是共享该变量的引用。浅克隆是简单也容易达成的，但是复杂结构原型的克隆一般需要进行深复制，因为这样可以保证原型对象与被克隆的对象彼此独立，互不影响。

虽然深克隆的问题比较复杂，但使用流化对象的技巧可以方便实现对象的深克隆。下面我给出一个演示性的例子。

图 8-7 是这个原型模式深克隆演示程序的类图。从图中看出，TStreamableClass 包含了 FContainedClass 和 FMemo 两个数据成员变量，分别引用了 TContainedClass 和 TMemoPrototype 的实例对象。TStreamableClass 的 Clone 方法实现了对象的深克隆。

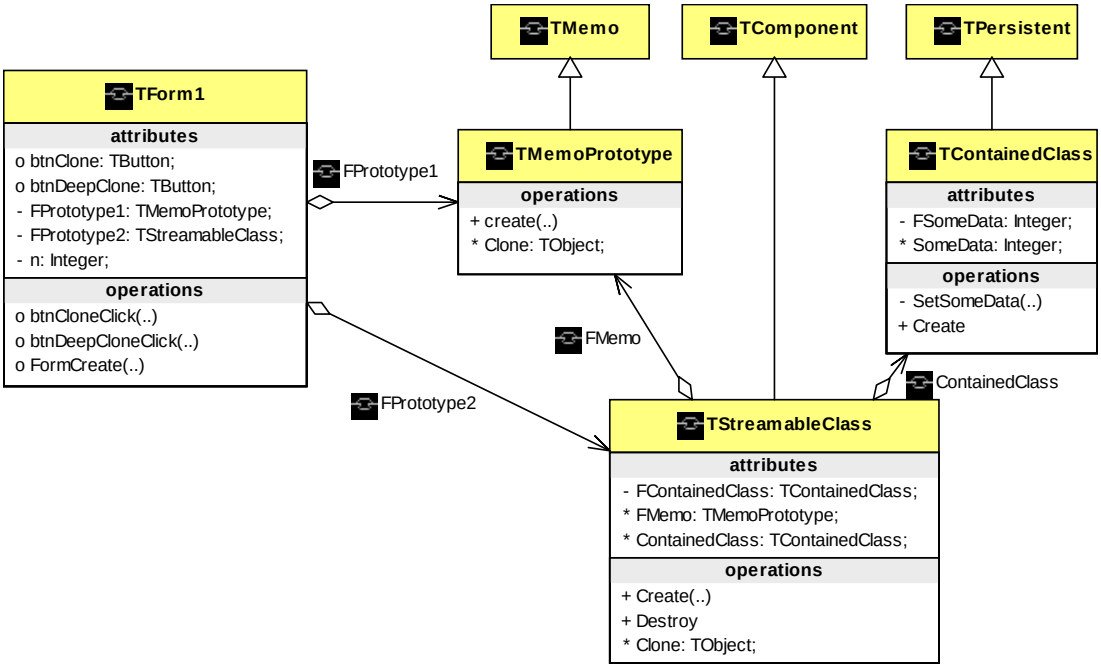


图 8-7 原型模式深克隆演示程序类图

示例程序 8-7 是 PrototypeByStream 单元程序的源码，这里实现了原型模式。需要注意的是，声明对象时，只有放置在 published 域的数据成员（或属性）才能够被自动流化。也就是说，要克隆的引用对象变量要么直接放在 published 域（如 TStreamableClass 的数据成员 FMemo），要么将访问其的属性放在 published 域（如 TStreamableClass 的属性 ContainedClass）。

从流中读出对象时，需要将该对象转型为原来的类型，比如：

```
AClassInstance :=
  TStreamableClass( ReadComponentResFile('DeepClone', nil));
```

但是事先要使用 RegisterClass 或 RegisterClasses 注册自己的类，比如：

```
RegisterClasses([TMemoPrototype,TContainedClass, TStreamableClass]);
```

否则会找不到类，出现如图 8-8 所示的 EClassNotFound 异常。

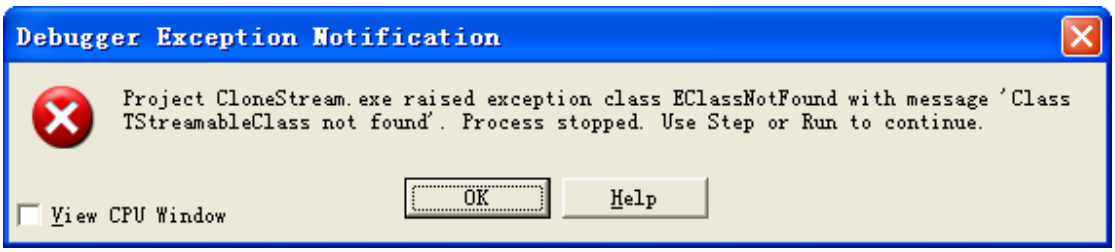


图 8-8 未注册类导致 EClassNotFound 异常

示例程序 8-7 PrototypeByStream 单元程序源码

```
unit PrototypeByStream;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs,StdCtrls;

type

  TMemoPrototype = class (TMemo)
  public
    constructor create(AOwner:TComponent);override;
  published
    function Clone: TObject;
  end;

  TContainedClass = class(TPersistent)
  private
    FSomeData: Integer;
    procedure SetSomeData(Value: Integer);
  public
    constructor Create;
  published
    property SomeData: Integer
             read FSomeData write SetSomeData;
  end;

  TStreamableClass = class(TComponent)
  private
    FContainedClass: TContainedClass;
  public
    constructor Create(AOwner: TComponent); override;
```

```
    destructor Destroy; override;
  published
    FMemo:TMemoPrototype;
    function Clone: TObject;
    property ContainedClass: TContainedClass
                 read FContainedClass write FContainedClass;
  end;

implementation

procedure TContainedClass.SetSomeData(Value: Integer);
begin
  FSomeData := Value;
end;

constructor TContainedClass.Create;
begin
  FSomeData := 42;
end;

constructor TStreamableClass.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FContainedClass := TContainedClass.Create;
  FMemo:=TMemoPrototype.create(AOwner);
end;

destructor TStreamableClass.Destroy;
begin
  FContainedClass.Free;
end;

function TStreamableClass.Clone: TObject;
var
  AClassInstance: TStreamableClass;
begin
  AClassInstance := TStreamableClass.Create(nil);
  WriteComponentResFile('DeepClone', AClassInstance);
  FreeAndNil(AClassInstance);
  AClassInstance :=
     TStreamableClass( ReadComponentResFile('DeepClone', nil));
  result:=AClassInstance;
end;
```

```
{TMemoPrototype}
function TMemoPrototype.Clone: TObject;
var
  tmp:TMemoPrototype;
  TmpStream:TStream;
begin
  WriteComponentResFile('MemoPrototype.dat',self);
  tmp:=TMemoPrototype(ReadComponentResFile('MemoPrototype.dat',nil));
  result:=tmp;
end;

constructor TMemoPrototype.create(AOwner:TComponent);
begin
  inherited;
  Width:=100;
  Height:=50;
  Left:=50;
  Top:=50;
  Font.Color:=clBlue;
end;

initialization
  RegisterClasses([TMemoPrototype,TContainedClass, TStreamableClass]);
finalization
end.
```

在示例程序 8-8 中，我使用并演示了使用流技术克隆对象的原型模式。除了在演示窗体上可以看到克隆的对象，我们还可以测试被克隆的引用对象是一个新对象还是原始对象的一个引用。该程序的运行界面如图 8-5 所示。

示例程序 8-8 演示窗体的程序源码

```
unit PrototypeStreamForm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls,PrototypeByStream;

type
  TForm1 = class(TForm)
    btnClone: TButton;
    btnDeepClone: TButton;
    procedure FormCreate(Sender: TObject);
```

```
    procedure btnCloneClick(Sender: TObject);
    procedure btnDeepCloneClick(Sender: TObject);
  private
    FPrototype1:TMemoPrototype;
    FPrototype2:TStreamableClass;
    n:integer;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  FPrototype1:=TMemoPrototype.create(self) ;
  n:=1;
end;

procedure TForm1.btnCloneClick(Sender: TObject);
var
  tmp:TMemoPrototype;
begin
  tmp:=TMemoPrototype(FPrototype1.clone);
  tmp.Name:='Memo'+IntToStr(n);
  tmp.Parent:=self;
  tmp.Lines.Add('克隆文本框之'+IntToStr(n));
  tmp.Left:=tmp.Left+30*n;
  tmp.Top:=tmp.Top+30*n;
  inc(n);
end;

procedure TForm1.btnDeepCloneClick(Sender: TObject);
var
  AInstance: TStreamableClass;
begin
  FPrototype2:=TStreamableClass.create(self);
  AInstance:=TStreamableClass(FPrototype2.Clone);
  showmessage(inttostr(AInstance.ContainedClass.SomeData));
  AInstance.ContainedClass.SomeData:=30;
```

```
  showmessage(inttostr(AInstance.ContainedClass.SomeData));
  AInstance.FMemo.Parent:=self;
  AInstance.FMemo.Lines.Add('Deep Clone OK!');

    //测试被克隆的引用对象是一个新对象还是原始对象的一个引用。
    if (AInstance.FMemo=FPrototype2.FMemo) or
       (AInstance.ContainedClass=FPrototype2.ContainedClass)
    then
    begin
      showmessage('克隆的是对象引用！');
    end;

end;

end.
```

这本书中，我介绍的重点是模式及其应用实践。而 Delphi 中关于流的技术和对象流化机制还相当复杂，有兴趣的读者可以进一步查阅有关书籍和资料。

8.3.4 范例小结

通过研究范例程序，我们不难体会出原型模式包括了以下优点：

- 原型模式允许动态地增加或减少产品类。由于创建产品类实例的方法是产品类内部具有的，因此增加新产品对整个结构没有影响。
- 原型模式提供简化的创建结构。工厂方法模式常常需要有一个与产品类等级结构相同的等级结构，而原型模式就不需要这样。
- 具有给一个应用软件动态加载新功能的能力。当需要增加一个新功能时，只需要提供一个新类的克隆，并在原型管理器中登记注册即可，而不必给每一个软件的用户提供一个全新的软件包。
- 产品类不需要非得有任何事先确定的等级结构，因为原型模式适用于任何的等级结构。

原型模式最主要的缺点是每一个类都必须配备一个克隆方法。配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类来说不是很难，而对于已经有的类不一定很容易。

第15章　代理模式（Proxy）

15.1　模式解说

所谓代理（Proxy）模式就是为某一个对象的访问提供一个代理对象，而不是直接去控制这个对象。这个代理对象在客户端和源对象之间起着中介的作用。

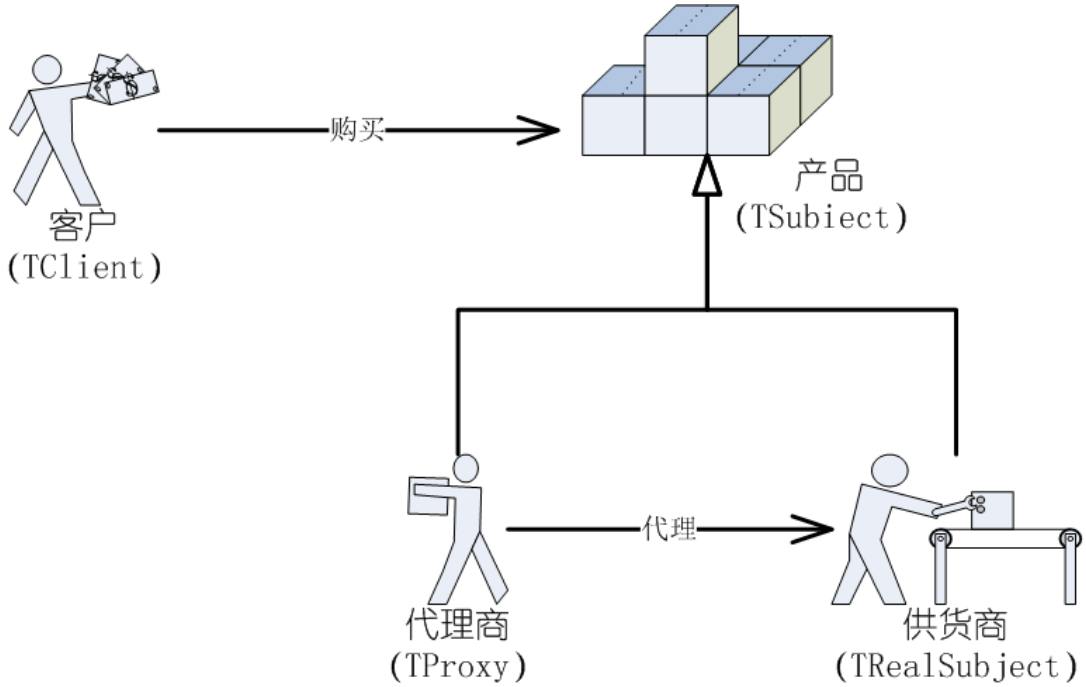


图 15-1 产品的代理模式

例如，有客户要购买某种专业产品，但这种产品仅有少数几家供货商生产。客户很难找到这些供货厂家；即使找到了，也可能遇到这些厂家缺货或停产的麻烦。如果客户通过代理商来购买这一产品，事情就变得简单，因为代理商知道哪些供货厂家有货源。于是代理商在客户对象和产品对象之间起着中介作用。

图 15-1 是产品代理模式的示意图。客户对象通过代理商购买产品，他所关心的不再是具体产品及其生产和供货的途径，而是抽象的产品，即作为接口的一份产品说明书。供货商是产品说明书的真正实现者，他严格按照产品说明书生产和提供符合要求的产品，代理商虽然也按照产品说明书的要求代理产品，但他的责任是根据产品说明书的要求选择合格的供货商来实现提供的产品，也就是说他自己并不具体实现该产品，他只是一个中介。由于有了代理商，客户不必知道那些供货商生产和提供产品，供货商也不必知那些客户需要产品，也就是说自从有了代理商，客户和供货商之间的依赖关系不复存在，代理商起到解耦作用。

在代理模式中，其核心包含接口、代理和实现三部分。接口抽象了客户感兴趣的操作，方便客户访问对象。代理和实现都是对接口的实现，但是其实现的方式不同。一个接口可能会有多种不同的具体实现（好比一种产品会有多个不同的供货厂家）；而代理可以通过选择和使用有效的具体实现来间接实现接口，为客户提供服务。所以说，代理并不仅仅是多了一道手续而是一项增值服务，代理通过自身的业务逻辑可以帮助具体实现解决问题、克服不足，

提升了服务质量。代理与实现的关系如图 15-2 所示。

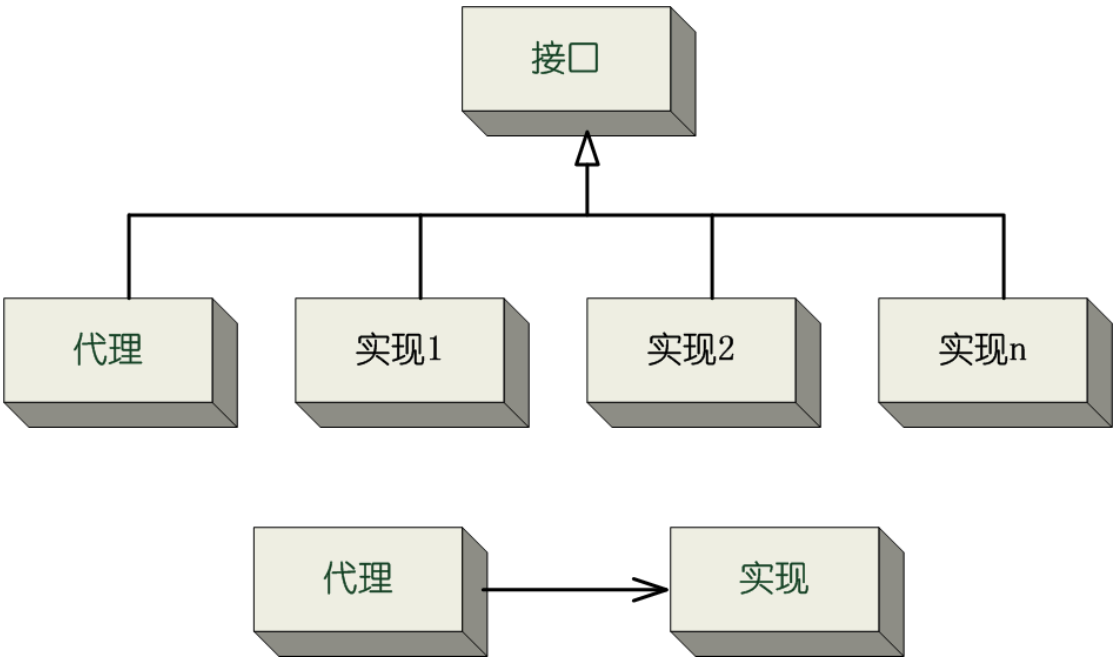


图 15-2 代理与实现的关系

在软件开发中，经常使用代理来解除对象与对象，系统与系统之间的强耦合，起到中介和缓冲的作用。

例如在如图 15-3 所示的看图软件设计中，如果图形浏览器一下子把所有图片文件都读到内存中，那么该图形浏览器的加载需要很长时间。因为有些图形对象的创建开销很大，造成了难以忍受的延时，因此设计图形浏览器时，应避免在加载时一次性创建所有的图形对象。实际上用户并非对所有的图形文件都感兴趣，所以也没有必要同时创建这些图形对象。

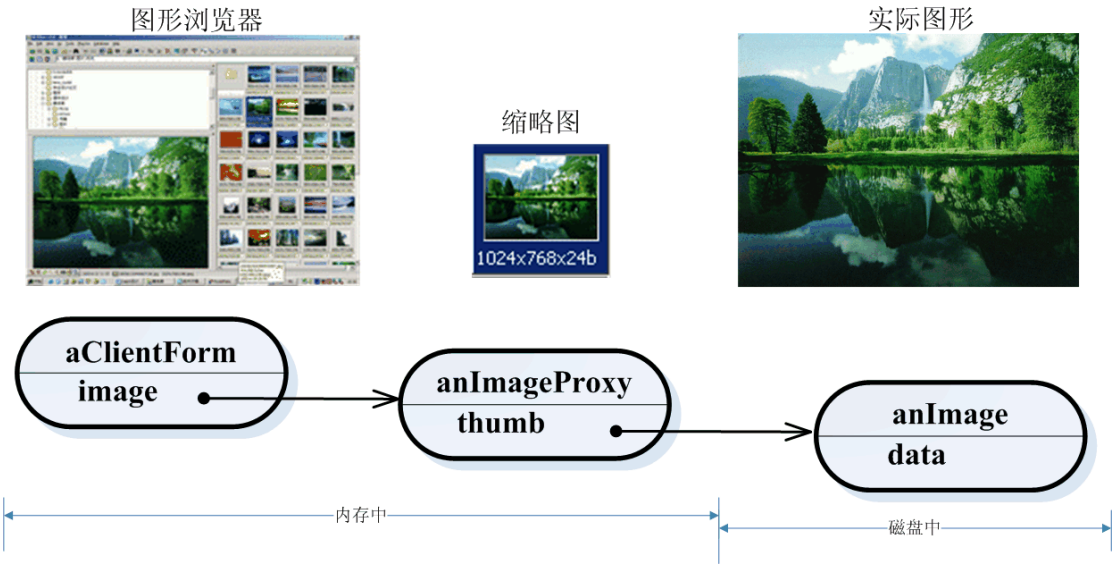


图 15-3 一个看图软件的代理模式

使用代理模式可以控制存取一个对象，特别是对于那些耗费资源的对象，代理模式可以将其延迟到真正需要使用时再创建。这就是说，图形浏览器加载时使用的不是真正的图形对象，而是另一个图形代理对象，即 ImageProxy。图形代理对象替代那个真正的图形对象，

并且在需要时负责实例化这个图形对象。图形代理对象用缩略图来代替实际的图形文件在图形浏览器中显示，仅当用户通过点击某个缩略图来激活图形代理对象显示操作时，该图形代理对象才加载图形文件并创建真正的图形。此时图形代理对象引用这个新建的图形对象，使用户能够完成对该图形的各种操作。

图形代理对象使用的缩略图一般保存在一个数据库中（例如：ACDSee 使用的 Thumbs.db 数据库）。缩略图在图形代理对象创建时加载并保存在内存中，而图形对象只在需要时创建，所以实际图形通常以图形文件形式保存在磁盘中。这样，图形代理对象缓解了图形浏览器与图形对象之间的依赖，起到中介和缓冲的作用。

代理是一个会经常用到的概念，而且种类繁多。可以使用代理模式的常见代理有：

- **远程代理（Remote Proxy）** 为一个对象在不同的地址空间提供局部代表对象。这个不同的地址空间可以是在本机中，也可以是在其他机器中。
- **虚代理（Virtual Proxy）** 根据需要创建开销很大的对象，使得该对象仅仅在实际使用时才会被真正创建出来。
- **保护代理（Protection Proxy）** 控制对原始对象的访问。保护代理可以提供或限制用户访问对象的不同权限。
- **智能引用（Smart Reference）** 当一个对象被引用时，智能指引取代了简单的指针。它提供了一些访问对象时的附加操作，比如：检查和管理引用计数，当引用为零时，负责自动释放该对象；检查引用对象是否锁定，确保不盲目改变对象的状态；检查并确保持久化对象是在首次加载时装入内存。

15.2 结构和用法

15.2.1 模式结构

代理模式的结构如图 15-4 所示，它包括了以下参与者：

- 代理（TProxy）——负责维护一个引用，使得代理可以访问真实主题对象；提供一个与主题一致的接口让代理可以替代真实的对象；控制对真实对象的访问，并可能负责创建和删除它；在把客户端的调用传递给真实主题之前或之后，执行某些辅助性操作。
- 抽象主题（TSubiect）——定义 TRealSubject 和 TProxy 的共同接口；这样就在任何使用 TRealSubject 的地方都可以使用 TProxy。
- 真实主题（TRealSubject）——定义代理对象可以代理的真实对象。

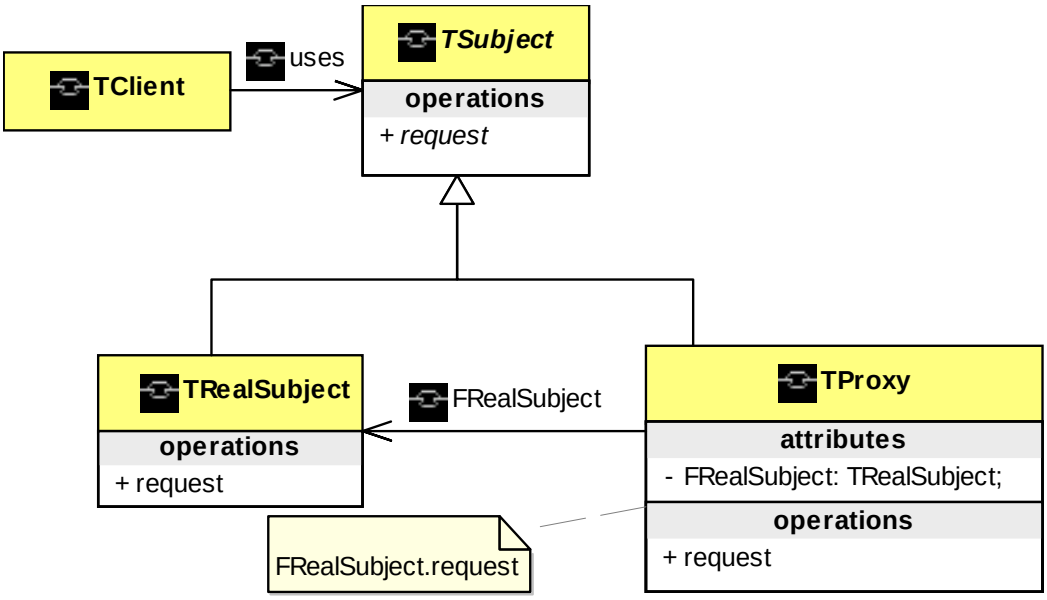


图 15-4 代理模式的结构

图 15-5 是在执行期的代理模式时序图。由图可见代理模式的工作过程为：客户端 Client 向代理对象 Proxy 发出请求，代理对象依据代理的种类适时传递请求给真实对象 RealSubject。这样，客户端对象、代理对象、真实主题对象之间就建立了如图 15-6 所示的引用关系。

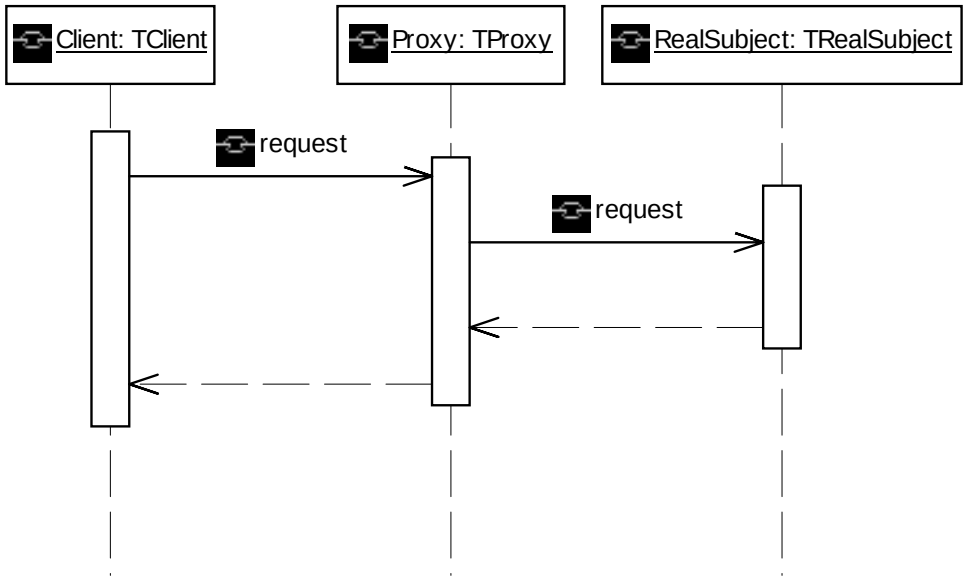


图 15-5 代理模式的时序图

由此可见，与客户端直接向真实对象发出请求的情况相比，代理可以通过间接传递请求而提供了比较灵活的中介控制。比如，代理可以在传递请求的前后执行一些附加操作，完成引用计数、权限控制、保护检查等操作；代理还可以决定何时、何地及如何创建或销毁所代理的真实对象。总之，代理模式将一个过渡层插入到客户端和真实主题之间，提供了一种游

刃有余的编程艺术。

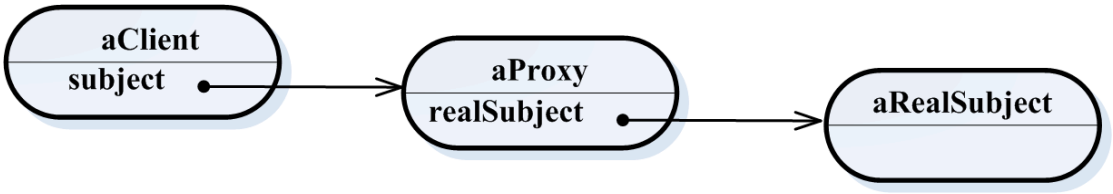


图 15-6 代理模式的对象引用关系

15.2.2 代码模板

示例程序 15-1 是实现代理模式的代码模板。从代码模板的实现上看并不复杂。其中，作为代理的 TProxy 类在实现作为主题的 TSubject 类的 request 接口时调用了被代理对象实现请求的方法。

```
procedure TProxy.request;
begin
  //调用被代理的对象实现请求方法
  if FRealSubject=nil then
     FRealSubject:=TRealSubject.Create;
  FRealSubject.request;
end;
```

但在实际应用中 TProxy 在 request 方法中还可以增加自己的处理代码，在传递请求的前后执行一些附加操作。

示例程序 15-1 代理模式的实现代码模板

```
unit Proxy;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TSubject = class (TObject)
  public
    procedure request; virtual; abstract;
  end;

  TRealSubject = class (TSubject)
  public
    procedure request; override;
```

```
  end;

  TProxy = class (TSubject)
  private
    FRealSubject: TRealSubject;
  public
    destructor destroy; override;
    procedure request; override;
  end;

implementation


{TProxy }
destructor TProxy.destroy;
begin
  FRealSubject.Free;
end;

procedure TProxy.request;
begin
  //调用被代理的对象实现请求方法
  if FRealSubject=nil then
      FRealSubject:=TRealSubject.Create;
  FRealSubject.request;
end;

{TRealSubject}
procedure TRealSubject.request;
begin
  //实现请求方法
  showmessage('这是 RealSubject 实现请求的方法!');
end;

end.
```

对客户端而言，原先要调用的真实主题对象可以被代理对象置换。因为他们实现的都是 TSubject 的共同的接口，所以在客户端看来并没有差别。这就是说，使用代理模式解决了问题，但丝毫不影响到客户端。下面是客户端使用代理模式模板单元的测试代码。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s:TSubject;
begin
```

```
  s:=TProxy.Create;
  s.request;
  s.Free;
end;
```

15.3 范例与实践

15.3.1 代理模式在数据库程序中的应用

1. 问题的提出

Smart PicLib 是我设计的一个基于数据库的图片管理系统，该程序的主要功能是访问数据库中的图片资料。该系统的运行界面如图 15-7 所示。

图 15-7 一个图片管理系统

虽然系统使用的数据库可以是本地的也可以是网络的，但由于从数据库中存取的是图片这样的长二进制数据，所以不良的设计可能会导致巨大的资源开销和难以忍受的运行等待。

这种说法绝不是我有意杜撰的。很多学习 Delphi 程序员都见过 Delphi 提供的如图 15-8 所示的示例程序（该程序位于 Delphi 安装目录下的\Demos\Db\FishFact）。他们为这种几乎不需要撰写多少代码的 RAD 开发方式着迷。但是我的一位读者朋友，成都的张先生曾经就用类似的方式编写了一个这样的图片管理程序。他使用的是微软 Access 数据库，当数据库中的图片数量超过 500 张（每张图片大小平均为 700k）时，程序的加载启动时间竟然达到不可思议的 5 分钟，而且操作运行速度也难以忍受。当他使用我的 Smart PicLib 进行同样条件下的测试（使用 Access 数据库，500 张 1200k 大小的图片），发现程序加载启动和操作运行几乎没有感觉到的延时。

所以看似简单的一个图片管理系统，在设计上还有许多值得思考和提高的地方。

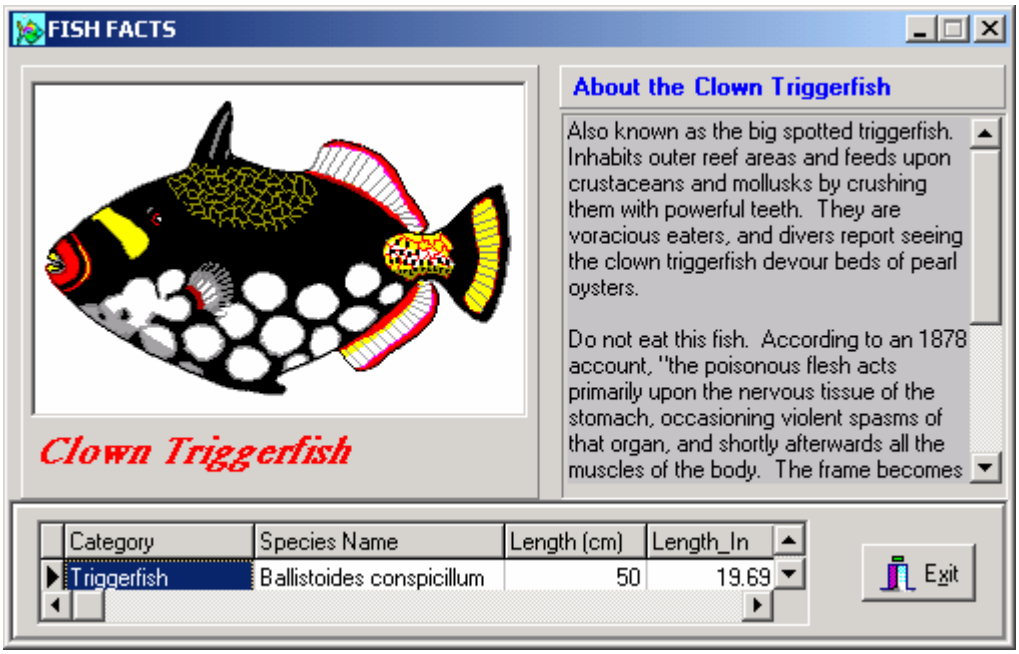


图 15-8Delphi 提供的一个访问数据库图片字段的范例程序

有经验的读者可能对此不以为然。他们知道张先生的问题出在使用 Delphi 的现成控件一下子把数据库中的记录都读进内存，造成了资源的紧张和运行的低效。

实际上一次从数据库打开 500 张图片对用户并没有太大的意义。如果采取一个巧妙的办法让用户先模糊查询一组感兴趣的图片名称，然后待用户选定（点击）一个图片名称后，再用 SQL 语句来访问这个图片。这样就可以解决程序启动和运行速度慢的问题。

这种做法的确可行，不过在具体实现中仍然有很多需要深入考虑的地方，比如：如何实现 O-R Mapping，即如何建立数据库和实体对象之间的关系？如何将业务逻辑与数据库操作分离？如何将系统的应用层从数据库 API 层解耦？

再详细一点。如果我们用一个 TPic 类的实例对象来对应（Mapping）数据库中的一条图片记录。假设这个 TPic 类声明如下：

```
TPic = class (TObject)
public
  ID: string;
  Name: string;
  Note: string;
  Pic: TPicture;
  PicDate: TDate;
  PicType: string;
  constructor Create;
  destructor Destroy; override;
  procedure XXX;
  ... ...
end;
```

那么当我们要删除图片时，会在查询列表中选择要删除的图片名称，以便进行删除操作。实际上我们要完成的删除操作包括对列表中的图片实体对象（即 TPic 类的实例对象）进行删除，以及对数据库中的图片记录进行删除，如示例程序 15-2 所示。

示例程序 15-2 删除图片

```
procedure TfrmPicView.actDeleteExecute(Sender: TObject);
var
  i:integer;
  Pic: TPic;
begin
  if application.MessageBox('确定要删除所选的图片吗？','操作提示',
       MB_OKCANCEL+MB_ICONEXCLAMATION)=1  then
  begin
    for  i:=clbName.Count-1  downto 0  do
    begin
      if clbName.Checked[i] then
      begin
        Pic:=TPic(clbName.Items.Objects[i]);
        //删除数据库中的图片记录
        with FADOQuery do
        begin
             SQL.Clear;
             SQL.Add('Delete from Pic ');
             SQL.Add(' where Id='+Pic.ID);
             ExecSQL;
            close;
        end;
        //删除图片实体对象
        Pic.free;
        //删除列表项
        clbName.Items.Delete(index);
      end;
    end;
  end;
end;
```

（注：这里的 clbName 是 TCheckListBox 类型）

这样程序代码中就会混合着界面逻辑（删除列表项）、业务逻辑（删除图片实体对象）与数据库操作逻辑（删除数据库中的图片记录），存在 3 个方向上的依赖关系。这样复杂的强耦合关系导致系统不以维护和扩展，没有弹性。

好在通过代理模式可以帮助我们优化设计，下面我就详细介绍基于代理模式优化设计和具体实现的图片管理系统 Smart PicLib。

2. 基于代理模式的优化设计

在设计时，我们都知道要把用户界面（User Interface）、业务（Business）和数据（Data）分层，程序员的水平高低决定了对这种分层的理解和实现。分层不仅仅是把实现逻辑代码分开，更重要的是合理设计他们的依赖关系，减低和简化彼此间的耦合。

例如在如图 15-9 所示的分层架构设计中，由于业务层和数据层关系纠缠不清，造成循环依赖；表现层同时依赖于业务层和数据层（例如示例程序 15-2），造成依赖关系复杂。这样虽然貌似分层，但实际上并未降低耦合度，达不到分层的真正效果。

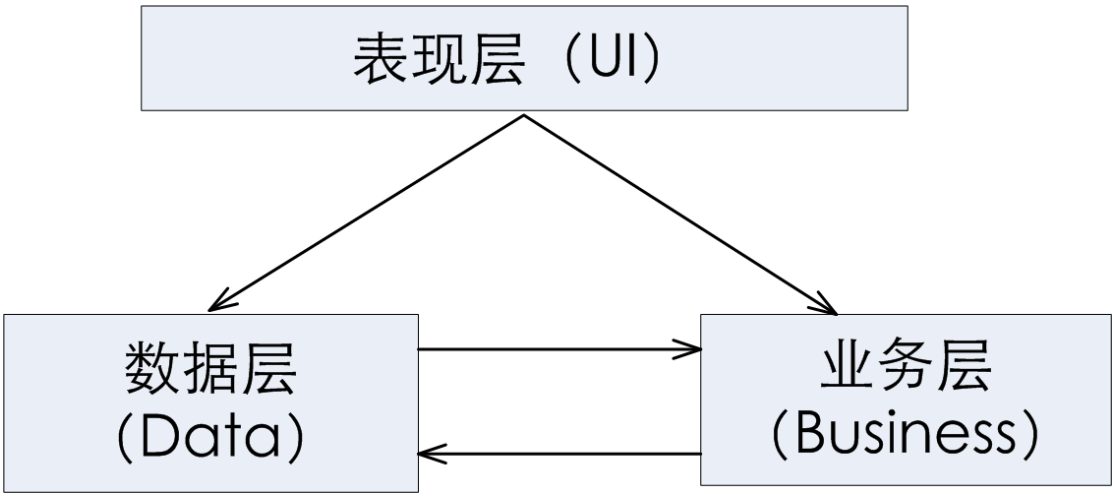


图 15-9 依赖关系混乱的分层

图 15-10 是单一依赖关系的分层，即表现层依赖于业务层，业务层依赖于数据层。这种分层的架构设计是我们经常使用的，它改进了图 15-9 所示的混乱的依赖关系，使得层与层之间的关系清晰、简单和易于维护。

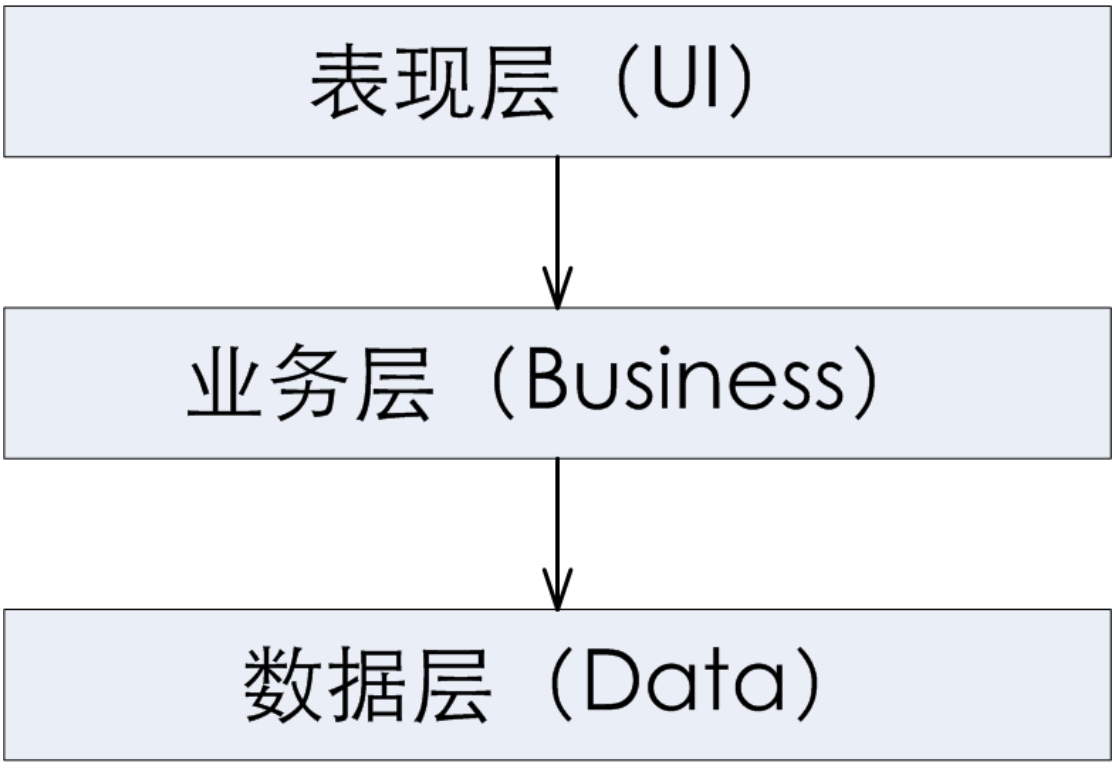


图 15-10 单一依赖关系的分层

在图 15-10 所示的层次关系中，表现层提供图形界面，让用户完成业务操作，所以表现层依赖于业务层是合理的。但是业务层对数据层的过分依赖并不是一件好事。因为我们可能会有不同的数据存储及管理需求以及基于不同平台的不同实现，比如：数据层可能使用 RDBMS 或 XML 等不同的持久机制，可能使用 ODBC、ADO、BDE、JDBC 等不同的数据访问接口（API）。为了适应这种变化，提高系统的健壮性（robustness），我们可以在业务层和数据层增加一层代理层，以吸收业务层和数据层的变化因素，消除他们之间的耦合，如图 15-11 所示。

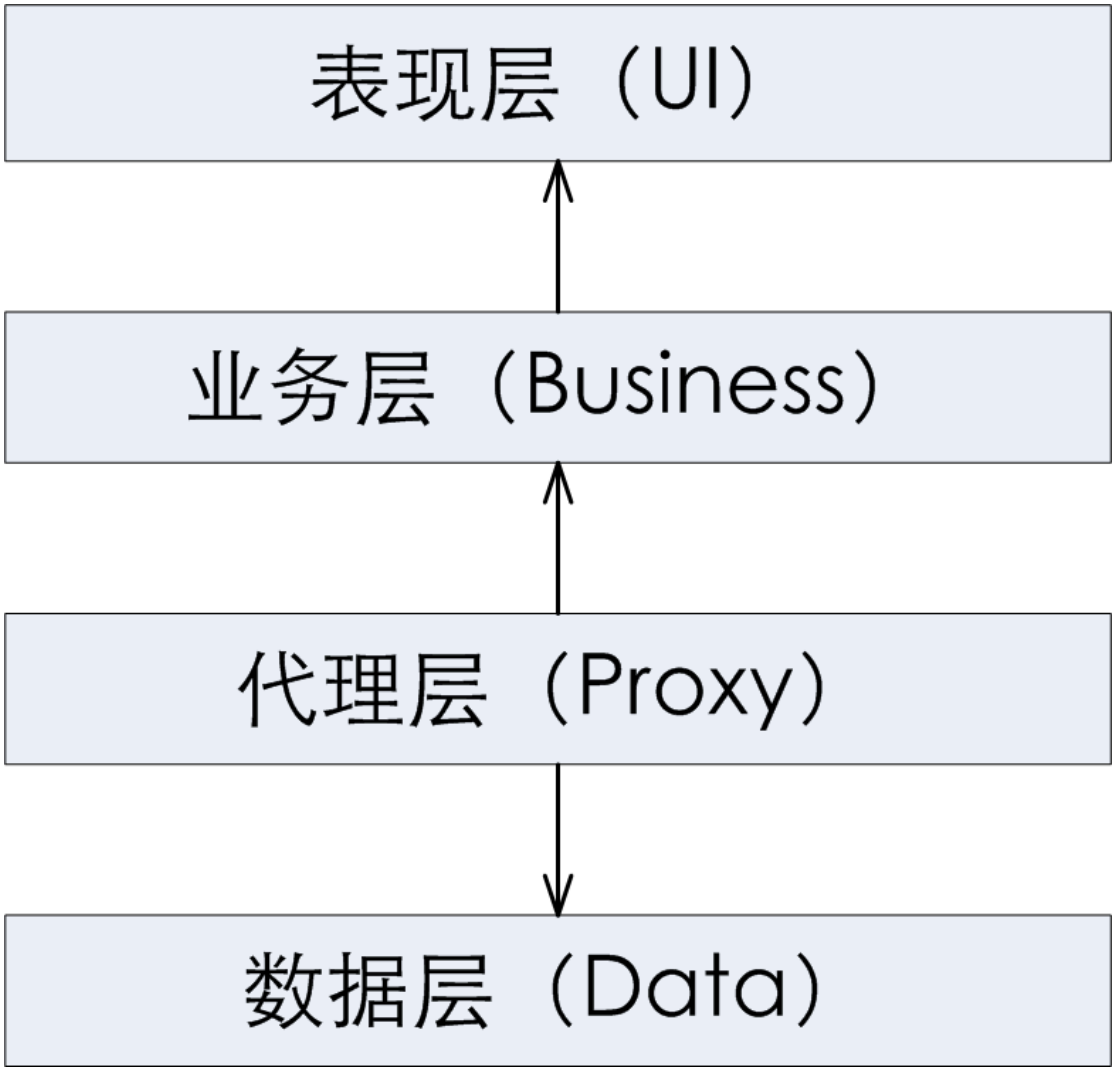


图 15-11 依赖反转的分层

注意图 15-11 所示的代理层分别依赖业务层和数据层，箭头方向不同于图 15-10 那样的顺序。这就是说代理层依赖业务层，而不是业务层依赖于代理层。如果是按照图 15-10 那样的依赖顺序加入代理层，即业务层依赖代理层，代理层依赖数据层，那么数据层的变化仍然有可能通过依赖它的代理层传播到业务层。我们通过依赖反转法则（The Dependency Inversion Principle，DIP）彻底隔绝了数据层对业务层的影响，解除了数据层和业务层的耦合。当然这里代理层的设计也非常有技巧，否则不但不能解决问题，反而为系统增添复杂性。

图 15-12 是基于代理模式对分层的设计。根据依赖反转法则“ 依赖抽象而不要依赖具体 ”的思想，我们可以把代理模式的抽象主题（Subiect）作为抽象接口让表现层和代理层依

赖，我们在真实主题（RealSubject）中实现业务逻辑，并在代理（Proxy）中完成业务层和数据层的胶合。这样一来，代理层就为业务层和数据层之间的 Mapping（包括但不限于 O-R Mapping）提供了集中统一的实现。

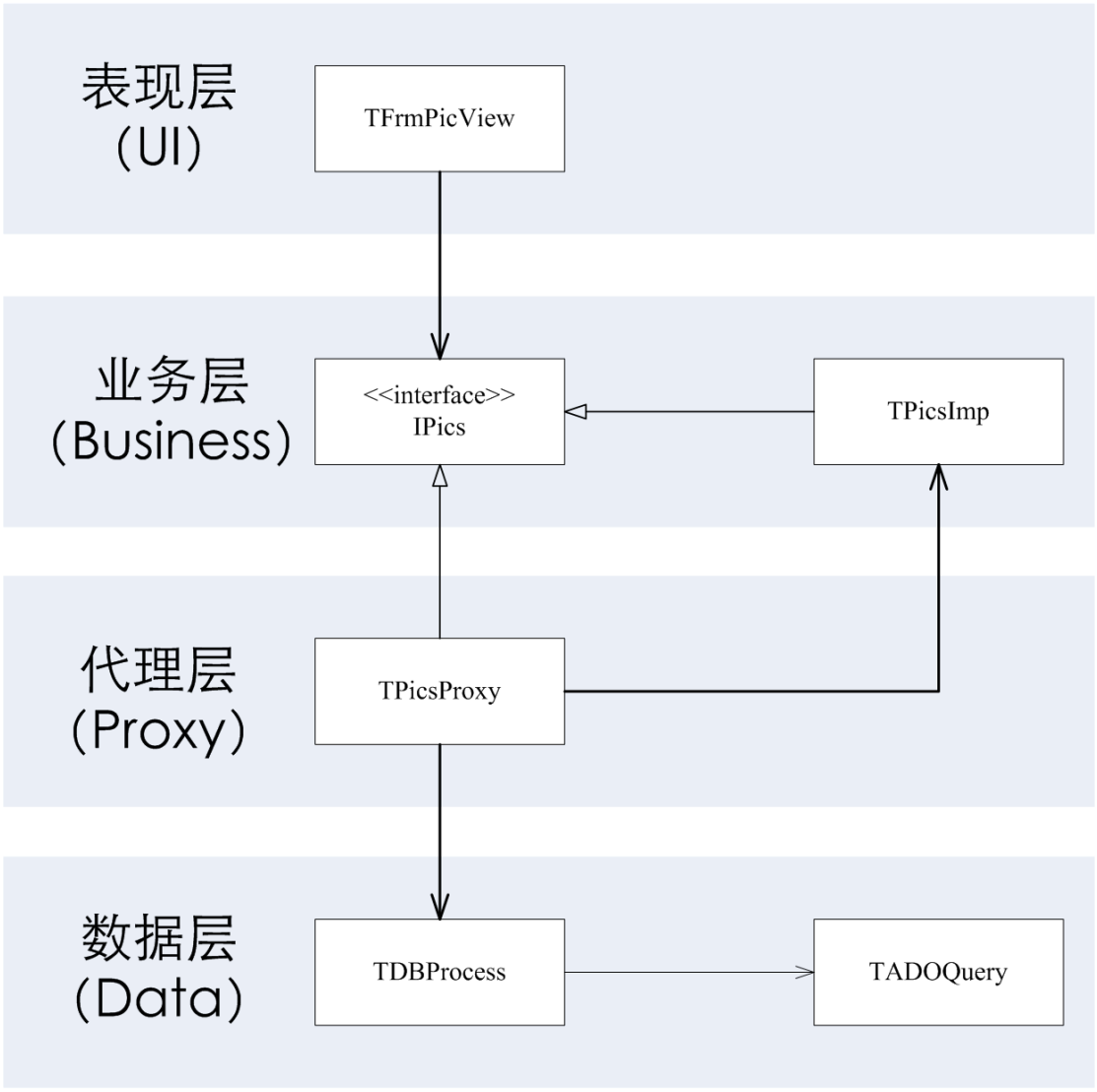


图 15-12 基于代理模式的设计可以由代理层吸收变化因素，增加了系统的健壮性

现在让我们回到前面我提到的那个基于数据库的图片管理系统 Smart PicLib，该程序的分层设计如图 15-12 所示，基于代理模式的具体设计如图 15-13 所示。

从图中我们看到，作为用户界面窗体的 TFrmPicView 负责表示层的逻辑，作为代理模式真实主题的 TPicsImp 负责业务层的逻辑，作为代理模式代理的 TPicsProxy 负责代理层的逻辑，作为数据层的 TDBProcess 负责数据层的逻辑。由于代理层依赖于业务层和数据层，并知道如何处理业务和数据之间的关系，所以它能起到中介和缓冲作用，从而避免了业务层和数据层之间的直接依赖和相互耦合。这就是说，业务层逻辑和数据层逻辑的变化不会导致他们之间的互相影响，而这些影响都作用到代理层了，并由代理层负责吸收和处理。显然，如果没有代理层，一但业务层逻辑和数据层逻辑发生变化，需要在业务层逻辑和数据层逻辑两处进行修改，而现在只需要在代理层一处进行修改，这种好处是不言而喻的。另外把业务逻辑和数据逻辑分离到两个不同的类中，也符合了单一职责法则（SRP），即一个类应该仅有一个原因导致其变化。否则业务和数据两重原因导致的变化会使系统难以维护和脆弱不

堪。

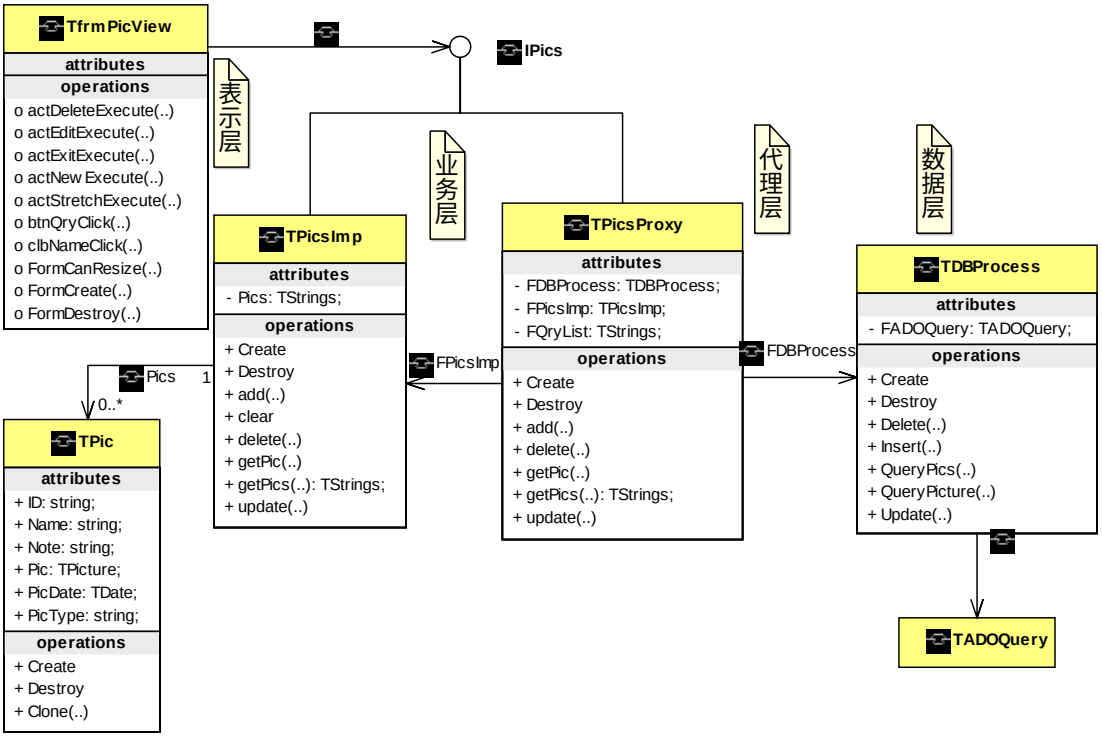


图 15-13 Smart PicLib 基于代理模式的设计

3. Smart PicLib 源码分析

PicProxy 单元的源代码如示例程序 15-3 所示，该单元实现了一个完整的代理模式。

首先我利用 Delphi 的对象接口作为代理模式的抽象主题。该接口声明了用于增加、更新、删除图片的操作。这里我说的图片既指图片的实体对象（TPic 的实例），也指数据库表中的图片记录。使用接口的好处就是可以通过不同的类对接口的不同实现，来分别对实体对象或数据记录进行操作。也就是说，具体实现层面上的操作变化不影响抽象表示层面上的接口定义。

```
IPics = interface (IInterface)
  ['{F514DB81-CE64-489E-9B1B-A60C4B63128C}']
  procedure add(Pic:TPic);
  procedure update(Pic:TPic;index:integer);
  procedure delete(index:integer;Pic:TPic);
  procedure getPic(PicID:string;var Pic:TPicture);
  function getPics(qryStr:string): TStrings;
end;
```

声明中的 getPics 操作，目的是通过模糊查询获得用户感兴趣的一组图片集（即图片对象列表）。但是该图片集并不立即返回图片的图像，只是返回图片的有关信息，包括图片名称，收集时间，内容描述等。待读者选中某一图片名称时再通过 getPic 操作返回实际的图片图像。这就是所谓虚代理（Virtual Proxy），即根据需要创建开销很大的对象，使得该对象仅仅在实际使用时才会被真正创建出来。

TPicsProxy 对 getPics 的实现如下：

```
function TPicsProxy.getPics(qryStr:string): TStrings;
var
  i:integer;
  pic:TPic;
begin
  //重新查询
  if qryStr<>'####' then
  begin
    FDBProcess.QueryPics(qryStr,FQryList);
    FPicsImp.clear;
    for i :=0  to FQryList.count-1 do
      FPicsImp.add(TPic(FQryList.objects[i]));
    end;
  result:=FPicsImp.getPics('') ;
end;
```

在使用 getPics 方法获得图片列表时存在两种情况，一种情况是用户定义新的条件进行重新查询，此时需要在数据库中完成新的查询操作；还有一种情况是用户对原有列表中的图片信息进行了修改，包括新增或删除，此时我们更新了数据库中的图片记录，但无需重新进行查询，只要刷新图片对象列表就可以了。所以我设计了一个标记“ #### ”，用于区分这两种情况。

尽量避免对数据库的不必要的操作是提高系统性能和效率的出发点，同样在读取图片图像时，我们也要这样去考虑。在 TPicsProxy 的 getPic 方法实现中，我们仅在用户首次浏览该图片图像时从数据库中获取该图片的图像，然后保存在图片实体对象中，如果用户再次浏览该图片图像，我们就直接显示图片列表中该图片对象所保存的图像，而不再访问数据库。其实这就是所谓的智能引用（Smart Reference），即当一个对象被引用时，智能指引取代了简单的指针。它提供了一些访问对象时的附加操作，确保不盲目改变对象的状态，检查并确保持久化对象是在首次加载时装入内存。我们这里的智能引用是引用类型的参数 Pic:TPicture，它来自于图片列表中该图片对象的数据成员 Pic。

```
procedure TPicsProxy.getPic(PicID:string;var Pic:TPicture);
var
  P:TPicture;
begin
  if Pic=nil then
  begin
    P:=TPicture.Create;
    FDBProcess.QueryPicture(PicID,p);
    Pic:=P;
  end;
end;
```

我们再来看看对图片的删除操作，比较一下和没有使用代理模式的示例程序 15-2 有何

不同。

在作为表示层的客户端程序中（参见示例程序 15-5），FPics 是一个 IPics 接口，它的创建如下：

```
FPics:=TPicsProxy.Create;
```

TfrmPicView 的 actDeleteExecute 实现了用户的删除操作，它调用了 FPics 的 delete 方法，用来删除列表框 clbName 中用户选定的图片：

```
procedure TfrmPicView.actDeleteExecute(Sender: TObject);
var
  i:integer;
begin
  if  trim(FPic.Name )='' then  exit;
  if application.MessageBox('确定要删除所选的图片吗？','操作提示',
       MB_OKCANCEL+MB_ICONEXCLAMATION)=1  then
  begin
    for  i:=clbName.Count-1  downto 0  do
    begin
      if clbName.Checked[i] then
        FPics.delete(i,TPic(clbName.Items.Objects[i]));
      end;
    clbName.Items:=FPics.getPics('####');
  end;
end;
```

由于 TPicsProxy 实现了 FPics 接口，所以该接口调用了 TPicsProxy 的 delete 方法：

```
procedure TPicsProxy.delete(index:integer;Pic:TPic);
begin
  FDBProcess.Delete(Pic);
  FPicsImp.delete(index,nil);
end;
```

显然，TPicsProxy 的 delete 方法中包含了数据层的删除逻辑和业务层的删除逻辑，他们实现删除的逻辑分别如下：

```
// 业务层的 TPicsImp 负责删除对象
procedure TPicsImp.delete(index:integer;Pic:TPic);
begin
  Pics.Objects[index].Free;
  Pics.Delete(index);
end;
```

```
//数据层的 TDBProcess 负责删除记录
procedure TDBProcess.Delete(Pic:TPic);
begin
  with FADOQuery do
  begin
    SQL.Clear;
    SQL.Add('Delete from Pic ');
    SQL.Add(' where Id='+Pic.ID);
    ExecSQL;
    close;
  end;
end;
```

代理层的 TPicsProxy 的 delete 方法通过对数据层对象 FDBProcess（TDBProcess 的实例）和业务层对象 FPicsImp（TPicsImp 的实例）的依赖，实现了整个应用程序的删除逻辑。其他操作的逻辑实现，如：新增、更新等与之相似，这里不再一一赘述。

在这里，我们看到数据层和业务层的逻辑是相互独立的，他们可以因各自的需求变化而独自演化。这种设计的好处是灵活、弹性，易于维护和重用。例如我要把数据层使用 ADO 数据库组件连接的 Access 数据库更换为用 dbExpress 数据库组件连接的 Oracle 数据库，那么对 TDBProcess 的改动与业务层的逻辑无关，同样在业务层 FPicsImp 中的任何改动也影响不到数据层。

示例程序 15-3 PicProxy 单元的源代码

```
unit PicProxy;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs,ExtCtrls,jpeg,dataModule;

type
  IPics = interface (IInterface)
    ['{F514DB81-CE64-489E-9B1B-A60C4B63128C}']
    procedure add(Pic:TPic);
    procedure update(Pic:TPic;index:integer);
    procedure delete(index:integer;Pic:TPic);
    procedure getPic(PicID:string;var Pic:TPicture);
    function getPics(qryStr:string): TStrings;
  end;

  TPicsImp = class (TInterfacedObject, IPics)
  private
    Pics: TStrings;
```

```
  public
    procedure clear;
    constructor Create;
    procedure add(Pic:TPic);
    procedure update(Pic:TPic;index:integer);
    procedure delete(index:integer;Pic:TPic);
    destructor Destroy; override;
    procedure getPic(PicID:string;var Pic:TPicture);
    function getPics(qryStr:string): TStrings;
  end;

  TPicsProxy = class (TInterfacedObject, IPics)
  private
    FPicsImp:TPicsImp;
    FDBProcess:TDBProcess;
    FQryList:TStrings;
  public
    constructor Create;
    destructor Destroy; override;
    procedure add(Pic:TPic);
    procedure update(Pic:TPic;index:integer);
    procedure delete(index:integer;Pic:TPic);
    procedure getPic(PicID:string;var Pic:TPicture);
    function getPics(qryStr:string): TStrings;
  end;

implementation


{TPicsImp }
constructor TPicsImp.Create;
begin
  Pics:=TStringList.Create;
end;

procedure TPicsImp.add(Pic:TPic);
begin
  Pics.AddObject(Pic.Name,Pic);
end;

procedure TPicsImp.update(Pic:TPic;index:integer);
var
  newPic:TPic;
begin
```

```
  if Pic.Pic=nil then
    Pic.Pic:=TPicture.Create;
  Pics.Strings[index]:=Pic.Name;
  newPic:=TPic(Pics.Objects[index]);
  newPic.Clone(Pic);
end;

procedure TPicsImp.delete(index:integer;Pic:TPic);
begin
  Pics.Objects[index].Free;
  Pics.Delete(index);
end;

destructor TPicsImp.Destroy;
begin
  clear;
  Pics.Free;
end;

function TPicsImp.getPics(qryStr:string): TStrings;
begin
  result:=Pics;
end;

procedure TPicsImp.getPic(PicID:string;var Pic:TPicture);
begin
  //
end;

procedure TPicsImp.clear;
var
  i:integer;
begin
  for  i:=Pics.Count-1 downto 0  do
    Pics.Objects[i].Free;
  Pics.Clear;
end;

{TPicsProxy }
constructor TPicsProxy.Create;
begin
  FPicsImp:=TPicsImp.Create;
  FDBProcess:=TDBProcess.create;
  FQryList:=TStringlist.Create;
```

```
end;

destructor TPicsProxy.Destroy;
begin
  FQryList.Free;
  FPicsImp.Free;
  FDBProcess.Free;
end;

procedure TPicsProxy.add(Pic:TPic);
var
  NewPic:TPic;
begin
  NewPic:=TPic.Create;
  NewPic.Clone(Pic);
  FDBProcess.Insert(NewPic);
  Pic.ID:=NewPic.ID;
  FPicsImp.add(NewPic);
end;

procedure TPicsProxy.update(Pic:TPic;index:integer);
begin
  FDBProcess.Update(Pic);
  FPicsImp.update(Pic,index);
end;

procedure TPicsProxy.delete(index:integer;Pic:TPic);
begin
  FDBProcess.Delete(Pic);
  FPicsImp.delete(index,nil);
end;

procedure TPicsProxy.getPic(PicID:string;var Pic:TPicture);
var
  P:TPicture;
begin
  if Pic=nil then
  begin
    P:=TPicture.Create;
    FDBProcess.QueryPicture(PicID,p);
    Pic:=P;
  end;
end;
```

```
function TPicsProxy.getPics(qryStr:string): TStrings;
var
  i:integer;
  pic:TPic;
begin
  //重新查询
  if qryStr<>'####' then
  begin
    FDBProcess.QueryPics(qryStr,FQryList);
    FPicsImp.clear;
    for i :=0  to FQryList.count-1 do
      FPicsImp.add(TPic(FQryList.objects[i]));
    end;
  result:=FPicsImp.getPics('') ;
end;

end.
```

示例程序 15-4 是数据层单元的源代码。TPic 是用来映射数据记录的，它的数据成员对应着数据表中的字段，同时它还提供了一个 Clone 方法用于复制对象。TDBProcess 集中了针对数据库操作的逻辑，包括：新增、更新、删除、查询操作。读者可以通过更改这里的实现代码，使这个 Smart PicLib 程序满足自己的数据库要求。

示例程序 15-4 dataModule 单元的源代码

```
unit dataModule;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs,ADODB,DB, DBClient;

type
  TPic = class (TObject)
  public
    ID: string;
    Name: string;
    Note: string;
    Pic: TPicture;
    PicDate: TDate;
    PicType: string;
    constructor Create;
    procedure Clone(p:TPic);
    destructor Destroy; override;
```

```
  end;

  TDBProcess = class (TObject)
  private
    FADOQuery: TADOQuery;
  public
    constructor Create;
    destructor Destroy;override;
    procedure Delete(Pic:TPic);
    procedure Insert(Pic:TPic);
    procedure QueryPics(qryStr:string;PicList:TStrings);
    procedure QueryPicture(PicID:string;Picture:TPicture);
    procedure Update(Pic:TPic);
  end;

const
  ADO_STRING='Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;'
    +'Data Source=MyPic.mdb; Mode=Share Deny None;'
    +'Extended Properties="";Jet OLEDB:System database="";'
    +'Jet OLEDB:Registry Path="";Jet OLEDB:Database Password="";'
    +'Jet OLEDB:Engine Type=5;'
    +'Jet OLEDB:Database Locking Mode=1;'
    +'Jet OLEDB:Global Partial Bulk Ops=2;'
    +'Jet OLEDB:Global Bulk Transactions=1;'
    +'Jet OLEDB:New Database Password="";'
    +'Jet OLEDB:Create System Database=False';

implementation

{TPic }
procedure TPic.Clone(p:TPic);
begin
  ID:=p.ID;
  Name:=p.Name;
  Note:=p.Note;
  PicDate:=p.PicDate;
  PicType:=p.PicType;
  Pic.Assign(p.Pic);
end;

constructor TPic.Create;
begin
  Pic:=TPicture.Create;
end;
```

```
destructor TPic.Destroy;
begin
  Pic.Free;
end;

{TDBProcess }
constructor TDBProcess.Create;
begin
//初始化数据库
  FADOQuery:=TADOQuery.Create(nil);
  FADOQuery.ConnectionString:=ADO_STRING;
end;

//删除记录
procedure TDBProcess.Delete(Pic:TPic);
begin
  with FADOQuery do
  begin
    SQL.Clear;
    SQL.Add('Delete from Pic ');
    SQL.Add(' where Id='+Pic.ID);
    ExecSQL;
    close;
  end;
end;

procedure TDBProcess.Insert(Pic:TPic);
var
  FileName:string;
begin
  FileName:='c:\tmp.'+Pic.PicType;
  Pic.Pic.SaveToFile(FileName);
  with FADOQuery do
  begin
    SQL.Clear;
    SQL.Add('Select Name,Note,PicType,PicDate,Pic from Pic ');
    SQL.Add(' where Id=0');
    open;
    Insert;
    FieldbyName('Name').AsString:=Pic.Name;
    FieldbyName('Note').AsString:=Pic.Note;
    FieldbyName('PicDate').AsDateTime:=Pic.PicDate;
    FieldbyName('PicType').AsString:=Pic.PicType;
```

```
    TBlobField(FieldbyName('Pic')).LoadFromFile(FileName);
    post;
    close;
    //由于 ID 字段是由数据库自动编号，所以这里要取回最新的 ID 编号
    SQL.Clear;
    SQL.Add('Select max(Id) as ID from Pic ');
    //SQL.Add(' where Id=0');
    open;
      Pic.ID:=FieldbyName('ID').AsString;
    close;
  end;
end;

//查询图片
procedure TDBProcess.QueryPicture(PicID:string;Picture:TPicture);
var
  FileName:string;
begin
  with FADOQuery do
  begin
    SQL.Clear;
    SQL.Add('Select Name,PicType,Pic from Pic ');
    SQL.Add(' where Id='+PicID);
    Open;
    if FieldByName('Pic').IsNull  then
      Picture:=nil
    else
    begin
      FileName:='tmp.'+FieldByName('PicType').AsString;
      TBlobField(FieldByName('Pic')).SaveToFile(FileName);
      Picture.LoadFromFile(FileName);
    end;
    close;
  end;
end;

//查询记录集
procedure TDBProcess.QueryPics(qryStr:string;PicList:TStrings);
var
  i:integer;
  pic:TPic;
begin
  //重新查询
  if qryStr<>'####' then
```

```
  begin
    PicList.clear;
    with FADOQuery do
    begin
      SQL.clear;
      //构建模糊查询 SQL 语句
      SQL.Add('Select ID,Name,PicDate,Note,PicType from Pic ');
      SQL.Add(' where Name like '+#39+'%'+qryStr+'%'+#39);
      SQL.Add('order by Name');
      Open;
      First;
      while not Eof do
      begin
        pic:=TPic.Create;
        pic.Pic:=nil;
        pic.ID:=FieldbyName('ID').AsString;
        pic.Name:=FieldbyName('Name').AsString;
        pic.Note:=FieldbyName('Note').AsString;
        pic.PicDate:=FieldbyName('PicDate').AsDateTime;
        pic.PicType:=FieldbyName('PicType').AsString;
        PicList.addobject(pic.Name,pic);
        next;
      end;
      close;
    end;
  end;
end;

//更新记录
procedure TDBProcess.Update(Pic:TPic);
var
  FileName:string;
begin
  FileName:='c:\tmp.'+Pic.PicType;
  Pic.Pic.SaveToFile(FileName);
  with FADOQuery do
  begin
    SQL.Clear;
    SQL.Add('Select Name,Note,PicType,PicDate,Pic from Pic ');
    SQL.Add(' where Id='+Pic.ID);
    open;
    Edit;
    FieldbyName('Name').AsString:=Pic.Name;
    FieldbyName('Note').AsString:=Pic.Note;
```

```
    FieldbyName('PicDate').AsDateTime:=Pic.PicDate;
    FieldbyName('PicType').AsString:=Pic.PicType;
    TBlobField(FieldbyName('Pic')).LoadFromFile(FileName);
    post;
    close;
  end;
end;

destructor TDBProcess.Destroy;
begin
  if FADOQuery.Active then FADOQuery.Close;
  FADOQuery.Free;
end;

end.
```

最后，我用 Delphi 强大的可视化 RAD 功能设计出 Smart PicLib 的界面如图 15-14 所示。

图 15-14 Smart PicLib 的界面设计

这里我使用了 TActionManager 组件来管理界面操作，并把操作代码置于各个 Action 中。调试通过的全部代码如示例程序 15-5 所示。

> 参见 TActionManager组件的使用方法参见拙作《Delphi6企业级解决方案及应用剖析》（机械工业出版社 2002年出版）第212页。

示例程序 15-5 mainform 单元的源代码

```
unit mainform;
```

```
interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, StdCtrls, CheckLst, Buttons, ComCtrls, ExtDlgs,
  PicProxy,jpeg,DateUtils,dataModule, ActnList, ToolWin, ActnMan,
  ActnCtrls, XPStyleActnCtrls, StdStyleActnCtrls, ImgList, Menus;

type
  TfrmPicView = class(TForm)
    Panel1: TPanel;
    clbName: TCheckListBox;
    Panel2: TPanel;
    PicStatus: TStatusBar;
    sbPic: TScrollBox;
    Image: TImage;
    ActionManager1: TActionManager;
    actNew: TAction;
    actEdit: TAction;
    actDelete: TAction;
    actStretch: TAction;
    actExit: TAction;
    ActionToolBar2: TActionToolBar;
    Panel3: TPanel;
    btnQry: TBitBtn;
    edtName: TLabeledEdit;
    ImageList1: TImageList;
    PopupMenu1: TPopupMenu;
    N1: TMenuItem;
    N2: TMenuItem;
    N3: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure btnQryClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure clbNameClick(Sender: TObject);
    procedure actNewExecute(Sender: TObject);
    procedure actEditExecute(Sender: TObject);
    procedure actDeleteExecute(Sender: TObject);
    procedure actStretchExecute(Sender: TObject);
    procedure actExitExecute(Sender: TObject);
    procedure FormCanResize(Sender: TObject; var NewWidth,
      NewHeight: Integer; var Resize: Boolean);
  private
    FPics:IPics;
```

```
    FPic:TPic;
    FQry:string;
    FNullPic:TPicture;
  public
    { Public declarations }
  end;

var
  frmPicView: TfrmPicView;

implementation

uses edtForm;

{$R *.dfm}

procedure TfrmPicView.FormCreate(Sender: TObject);
begin
  FPics:=TPicsProxy.Create;
  FPic:=TPic.Create;
  FNullPic:=TPicture.Create;
  FNullPic.Assign(Image.Picture);
end;

procedure TfrmPicView.btnQryClick(Sender: TObject);
begin
  //查询并同步更新列表框（从数据库同步）
  clbName.Items:=FPics.getPics(edtName.Text);
  if  clbName.Items.Count>0 then
    clbName.ItemIndex:=0
  else
    Image.Picture.Assign(FNullPic);
end;

procedure TfrmPicView.FormDestroy(Sender: TObject);
begin
  FPics:=nil;
  FPic.Free;
  FNullPic.Free;
end;

//查看
procedure TfrmPicView.clbNameClick(Sender: TObject);
var
```

```
  tmpPic:TPic;
begin
  tmpPic:=TPic(clbName.Items.Objects[clbName.ItemIndex]);
  FPics.getPic(tmpPic.ID,tmpPic.Pic);
  Image.Picture.Assign(tmpPic.Pic);
  FPic.Clone(tmpPic);
  PicStatus.Panels.Items[0].Text:=DateTimeToStr(FPic.PicDate);
  PicStatus.Panels.Items[1].Text:=FPic.Note;
  frmEdit.edtName.Text:=FPic.Name;
  frmEdit.edtNote.Text:=FPic.Note;
  frmEdit.edtType.Text:=FPic.PicType;
  frmEdit.dtpPic.DateTime:=FPic.PicDate;
  frmEdit.imgPic.Picture.Assign(FPic.Pic);
  Image.Stretch:=True;
end;

//新增图片
procedure TfrmPicView.actNewExecute(Sender: TObject);
begin
  //初始化设置
  frmEdit.edtName.Text:='新图片';
  frmEdit.edtNote.Text:='';
  frmEdit.edtType.Text:='jpg';
  frmEdit.dtpPic.DateTime:=Today;
  frmEdit.imgPic.Picture.Assign(FNullPic);
  frmEdit.ShowModal;
  //保存
  if  frmEdit.ModalResult=mrOK then
  begin
    Image.Picture.Assign(frmEdit.imgPic.Picture);
    FPic.Name:=frmEdit.edtName.Text;
    FPic.Note:=frmEdit.edtNote.Text;
    FPic.PicType:=frmEdit.edtType.Text;
    FPic.PicDate:=frmEdit.dtpPic.DateTime;
    FPic.Pic.Assign(frmEdit.imgPic.Picture);
    if  (trim(FPic.Name)<>'')  then
      FPics.add(FPic);
  end;
  clbName.Items:=FPics.getPics('####');
end;

//编辑图片
procedure TfrmPicView.actEditExecute(Sender: TObject);
begin
```

```
  if  trim(FPic.Name )='' then  exit;
  frmEdit.edtName.Text:=FPic.Name;
  frmEdit.edtNote.Text:=FPic.Note;
  frmEdit.edtType.Text:=FPic.PicType;
  frmEdit.dtpPic.DateTime:=FPic.PicDate;
  frmEdit.ShowModal;
  //保存
  if  frmEdit.ModalResult=mrOK then
  begin
    FPic.Name:=frmEdit.edtName.Text;
    FPic.Note:=frmEdit.edtNote.Text;
    FPic.PicType:=frmEdit.edtType.Text;
    FPic.PicDate:=frmEdit.dtpPic.DateTime;
    FPic.Pic.Assign(frmEdit.imgPic.Picture);
    FPics.update(FPic,clbName.ItemIndex);
    Image.Picture.Assign(FPic.Pic);
  end;
  //同步更新列表框（不从数据库同步）
  clbName.Items:=FPics.getPics('####');
end;

procedure TfrmPicView.actDeleteExecute(Sender: TObject);
var
  i:integer;
begin
  if  trim(FPic.Name )='' then  exit;
  if application.MessageBox('确定要删除所选的图片吗？','操作提示',
       MB_OKCANCEL+MB_ICONEXCLAMATION)=1  then
  begin
    for  i:=clbName.Count-1  downto 0  do
    begin
      if clbName.Checked[i] then
        FPics.delete(i,TPic(clbName.Items.Objects[i]));
      end;
    clbName.Items:=FPics.getPics('####');
  end;
end;

procedure TfrmPicView.actStretchExecute(Sender: TObject);
begin
//通过单击图片可以切换不同的显示方式
  if not Image.Stretch  then
  begin
    Image.AutoSize:=False;
```

```
    Image.Align:=alNone;
    Image.Width:=Panel2.Width-6;
    Image.Height:=Panel2.Height-6;
    sbPic.HorzScrollBar.Visible:=False;
    sbPic.VertScrollBar.Visible:=False;
  end
  else
  begin
    Image.AutoSize:=True;
    Image.Align:=alClient;
    sbPic.HorzScrollBar.Range:=Image.Picture.Width;
    sbPic.VertScrollBar.Range:=Image.Picture.Height;
    sbPic.HorzScrollBar.Visible:=True;
    sbPic.VertScrollBar.Visible:=True;
  end;
  Image.Stretch:=not Image.Stretch;
end;

procedure TfrmPicView.actExitExecute(Sender: TObject);
begin
  close;
end;

procedure TfrmPicView.FormCanResize(Sender: TObject; var NewWidth,
  NewHeight: Integer; var Resize: Boolean);
begin
  if  not Image.AutoSize then
  begin
    Image.Width:=Panel2.Width-6;
    Image.Height:=Panel2.Height-6;
  end;
end;

end.
```

另外，主窗体 TfrmPicView 中还调用到一个用于编辑图片信息的模态窗体 TfrmEdit，该窗体的设计如图 15-15 所示，其实现代码如示例程序 15-6 所示。

示例程序 15-6 edtForm 单元的源代码

```
unit edtForm;

interface

uses
```

```
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, StdCtrls, ComCtrls, ExtDlgs,StrUtils;

type
  TfrmEdit = class(TForm)
    edtName: TLabeledEdit;
    edtNote: TLabeledEdit;
    edtType: TLabeledEdit;
    imgPic: TImage;
    dtpPic: TDateTimePicker;
    Label1: TLabel;
    btnPic: TButton;
    btnOK: TButton;
    OpenPictureDialog: TOpenPictureDialog;
    btnSave: TButton;
    procedure btnPicClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmEdit: TfrmEdit;

implementation

{$R *.dfm}

procedure TfrmEdit.btnPicClick(Sender: TObject);
var
  i:integer;
begin
  if OpenPictureDialog.Execute then
  begin
    //载入图片文件
    imgPic.Picture.LoadFromFile(OpenPictureDialog.FileName);
    //获取图片文件的扩展名
    i:=Pos('.',OpenPictureDialog.FileName);
    edtType.Text:=RightStr(OpenPictureDialog.FileName,
     (Length(OpenPictureDialog.FileName)-i));
  end;
end;
```

```
end.
```

图 15-15 用于编辑图片信息的窗体界面设计

15.3.2 范例小结

通过范例程序，我们对使用代理模式的意义和使用效果有了更深刻的体会，可以小结如下：

代理模式使得对对象的直接访问变成了间接代理访问。根据代理的不同类型，这种间接性带来诸多好处，例如：虚拟代理（Virtual Proxy）可以进行效率优化，根据要求创建对象；智能引用（Smart Reference）提供了一些访问对象时的附加操作，确保不盲目改变对象的状态。

在系统分层架构中，通过代理模式引入的代理层，可以使得原来互相依赖的层之间实现依赖反转，从而满足层与层之间独立演化的需求。特别在业务层依赖数据层或其他第三方中间层时，这种设计可以很好地消除耦合。

第24章　策略模式（Strategy）

24.1　　模式解说

策略（Strategy）模式的用意是定义一组算法（algorithms），并将每个算法封装到具有共同接口的独立的类中，从而使它们可以相互替换。策略模式让算法变化独立于使用它的客户端。

要了解策略模式的使用动机和意义，我们得先从一个有趣的例子说起。在一个物料管理系统中，出库和入库模块是该系统的核心部分（下面我们以出库为例进行分析）。

对于一个没有面向对象编程经验的程序员，他们往往会把出库的所有逻辑都放在客户端（出库单界面），并在客户端利用条件分支语句来判断该出库单类型是领料、借料还是报损，以便选择不同的出库结算方法，如图 24-1 所示。这样一来，客户端的代码就变得复杂和难以维护。比如：需要新增调拨单类型的出库时，就要修改判断条件，重新编译和发布客户端。当情况愈来愈复杂，条件分支会愈来愈多，添加的程序代码也会愈来愈多，这样让客户端愈来愈大并难以维护，互相影响和出错的可能性增大。

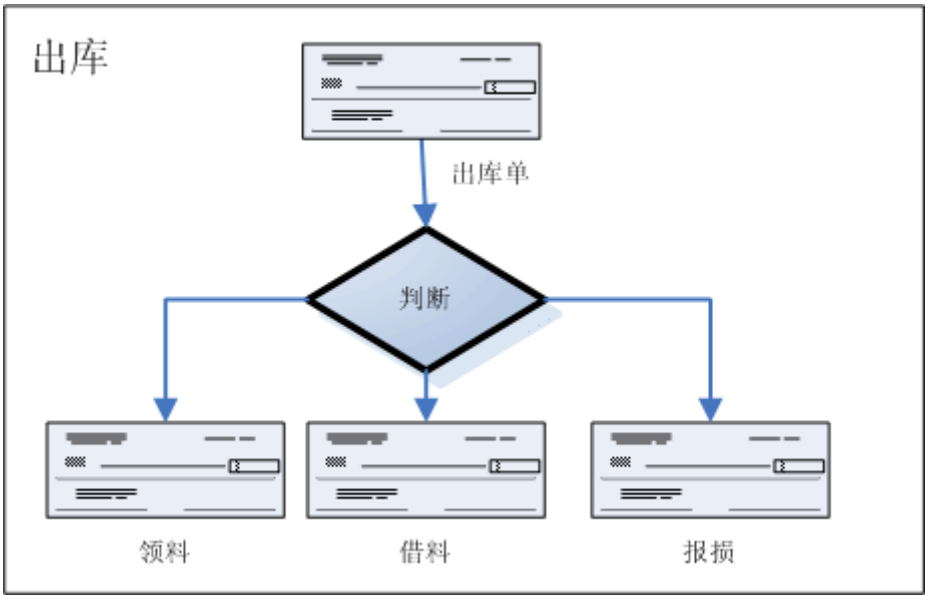


图 24-1 基于面向过程思想设计的出库模块

如果用面向对象的思想来分析，可以把领料单、借料单、报损单看作是出库单的派生类，如图 24-2 所示。这样出库单作为单据基类提供单据的共同接口，而利用继承的办法在子类里实现不同的出库行为。这实际上利用了面向对象里的一个重要概念：多态。

但是这样的设计还有美中不足的地方，这就是环境和行为紧密耦合在一起。也就是说，单据和具体出库的算法紧密耦合在一起。强耦合使得两者不能独立演化，限制了重用性和扩展性。

图 24-3 是利用策略模式重新设计的出库模块。出库单据对象通过一个出库操作对象（即

策略模式中的 Context）来引用出库策略对象。各种具体的出库策略则由出库策略类的派生类实现。出库单据可以由出库操作和单据样式分别提供出库结算方法和单据显示界面。这样，策略模式就把出库的行为从出库单据的环境中独立出来，出库算法的增减、修改都不会影响到环境和客户端。

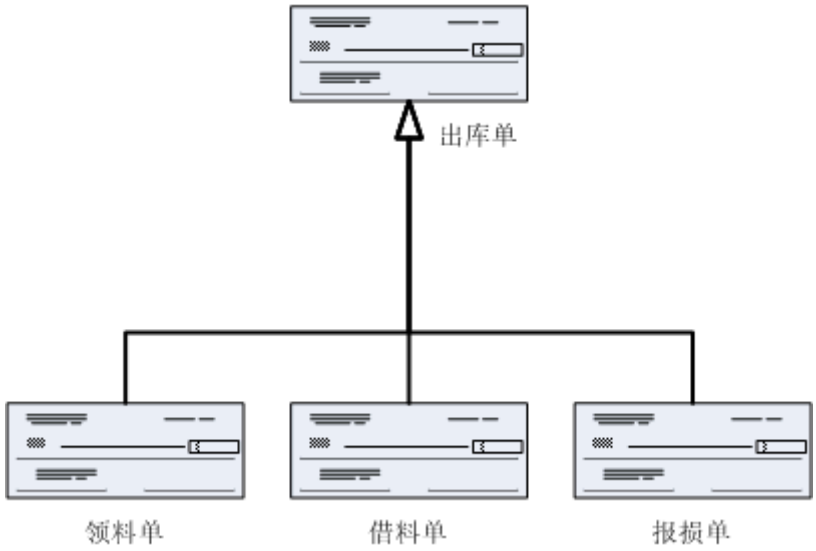


图 24-2 基于面向对象思想设计的出库模块

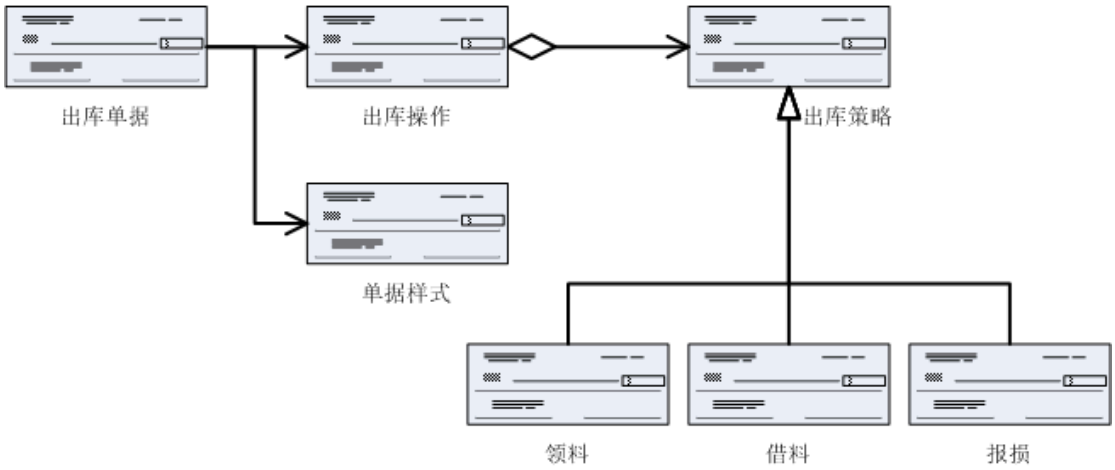


图 24-3 基于设计模式思想设计的出库模块

策略模式的优势在于算法和环境的分离，两者可以独立演化。为了更好地说明算法和环境分离的好处，我们不妨看一下图 24-4 的设计。在这个设计中，已经没有出库和入库模块的概念，因为我将所有出/入库单据抽象出来，在运行期动态组合单据的界面和行为。通过出/入库操作类，可以维护、查询、配置不同的行为类。抽象出的出/入库行为以策略类的方式封装了其对应的算法，以便完成不同类型的出入库单据的操作。这就显而易见地提高了系统的重用性和可扩展性，减低维护的难度。

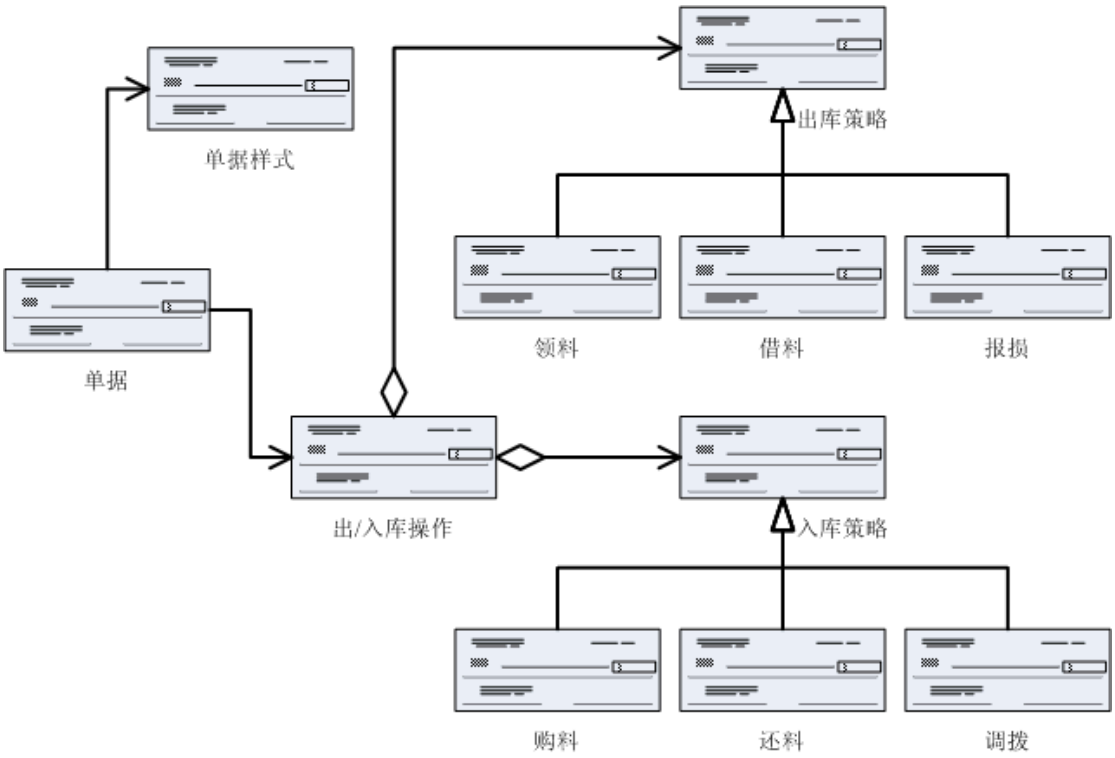


图 24-4 策略模式的优势在于算法和环境的分离，两者可以独立演化

由此可见，策略模式适用于以下情形：

- 当许多相关的类之间的差异只在于其行为时。策略模式可以动态地让一个对象在许多行为中选择一种行为。
- 当实现一个目的有多种可选算法时，比如：你出于不同的利弊权衡考虑定义的那些算法（即相当于应用不同的策略）。这些具体的算法可以封装成抽象算法类的派生类，并享用该抽象算法类的统一接口。通过多态性，客户端只要持有一个抽象算法类的对象，就可以选用任何一个具体的算法。
- 当一个算法使用的数据不可以让客户端得知时。使用策略模式可以避免暴露复杂的与算法相关的数据结构。其实客户端也没有必要知道这些与算法相关的知识和数据。
- 当一个类定义有很多行为，且用多个条件语句来判断选择这些行为时。策略模式可以把这些行为转移到对应的具体策略类中，从而避免了难以维护的多重条件选择，体现了面向对象的编程思想。

24.2 结构与用法

24.2.1 模式结构

策略模式的结构如图 24-5 所示，它包括了以下参与者：

- 抽象策略（TStrategy）——为所有支持的算法声明一个共同的接口。TContext 使用这个接口调用由 TConcreteStrategy 定义和封装的算法。

- 具体策略（TConcreteStrategy）——封装了具体算法或行为。实现 TStrategy 接口。
- 上下文（TContext）——持有一个到 TStrategy 的引用。调用 TStrategy 接口，动态配置具体算法或行为。

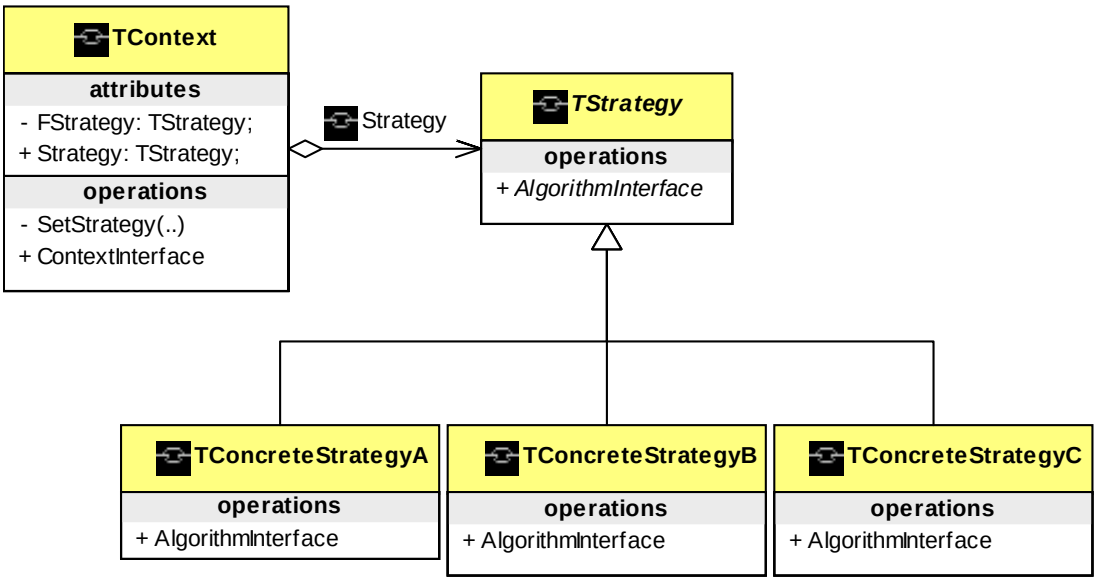


图 24-5 策略模式的结构

在策略模式中，通过 TStrategy 和 TContext 的交互实现所选择的算法。当算法被调用时，TContext 可以将该算法所需要的所有数据都传递给该 TStrategy。或者，TContext 可以将自身作为一个参数传递给 TStrategy 操作。

当 TContext 将客户端请求转发给它的 TStrategy 时，客户通常创建并传递一个 TConcreteStrategy 对象给该 TContext；这样，客户端仅与 TContext 交互。通常有一系列的 TConcreteStrategy 类可供客户端从中选择。

24.2.2 代码模板

示例程序 24-1 是实现策略模式的代码模板。该模板中，我示意性地给出了 A、B、C 三个具体策略。TContext 通过 Strategy 属性，持有了一个对 TStrategy 策略对象的引用，以便通过自己的 ContextInterface 方法访问 TStrategy 的 AlgorithmInterface 抽象接口。

示例程序 24-1 策略模式实现代码模板

```
unit Strategy;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
```

```
  TStrategy = class (TObject)
  public
    procedure AlgorithmInterface; virtual; abstract;
  end;

  TConcreteStrategyA = class (TStrategy)
  public
    procedure AlgorithmInterface; override;
  end;

  TConcreteStrategyB = class (TStrategy)
  public
    procedure AlgorithmInterface; override;
  end;

  TConcreteStrategyC = class (TStrategy)
  public
    procedure AlgorithmInterface; override;
  end;

  TContext = class (TObject)
  private
    FStrategy: TStrategy;
    procedure SetStrategy(Value: TStrategy);
  public
    procedure ContextInterface;
    property Strategy: TStrategy read FStrategy write SetStrategy;
  end;

implementation

{TConcreteStrategyA}
procedure TConcreteStrategyA.AlgorithmInterface;
begin
  //策略方法 A
  showmessage('执行策略方法 A');
end;

{TContreteStrategyB}
procedure TConcreteStrategyB.AlgorithmInterface;
begin
  //策略方法 B
  showmessage('执行策略方法 B');
end;
```

```
{TContreteStrategyC }
procedure TConcreteStrategyC.AlgorithmInterface;
begin
  //策略方法 C
  showmessage('执行策略方法 C');
end;

{TContext}
procedure TContext.ContextInterface;
begin
  Strategy.AlgorithmInterface;
end;

procedure TContext.SetStrategy(Value: TStrategy);
begin
  FStrategy:=Value;
end;



end.
```

为了便于读者了解客户端是如何使用策略模式的，我给出了一个简单的客户端示例，如示例程序 24-2 所示。该示例程序演示了用户通过界面选项，动态选择策略（算法）的效果。

示例程序 24-2 客户端程序使用策略模式的示例代码

```
unit ClientForm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls,Strategy;

type
  TClient = class(TForm)
    RadioGroup1: TRadioGroup;
    Button1: TButton;
    Button2: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
```

```
    FStrategyA:TStrategy;
    FStrategyB:TStrategy;
    FStrategyC:TStrategy;
    FContext:TContext;
  public
    { Public declarations }
  end;

var
  Client: TClient;

implementation


{$R *.dfm}

procedure TClient.FormCreate(Sender: TObject);
begin
  FStrategyA:=TConcreteStrategyA.Create;
  FStrategyB:=TConcreteStrategyB.Create;
  FStrategyC:=TConcreteStrategyC.Create;
  FContext:=TContext.Create;
end;

procedure TClient.Button1Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0:FContext.Strategy:=FStrategyA ;
    1:FContext.Strategy:=FStrategyB ;
    2:FContext.Strategy:=FStrategyC  ;
  end;
  FContext.ContextInterface;
end;

procedure TClient.FormDestroy(Sender: TObject);
begin
  FContext.Free;
  FStrategyA.Free;
  FStrategyB.Free;
  FStrategyC.Free;
end;

procedure TClient.Button2Click(Sender: TObject);
begin
```

```
  close;
end;

end.
```

24.2.3 问题讨论

1. TStrategy 及 TContext 接口的参数传递

TStrategy 和 TContext 接口必须使得 TConcreteStrategy 能够有效地访问它所需要的 TContext 中的任何数据，反之亦然。Delphi 有两种方法可以实现 TStrategy 和 TContext 接口的参数传递。

一种办法是让 TContext 将数据放在参数中传递给 TStrategy 操作。也就是说，将数据发送给 TStrategy。这种方法的优点是，使得 TStrategy 和 TContext 解耦。缺点是，为了兼顾不同的 TConcreteStrategy 需要，TContext 可能发送一些 TStrategy 不需要的数据。为此，在设计参数时，我们需要一些技巧。在后面的范例中，你会发现我使用了一些小花招来解决这个问题。

另一种办法是让 TContext 将自身作为一个参数传递给 TStrategy， 该 TStrategy 再显式地向该 TContext 请求数据。或者，TStrategy 可以存储对它的 TContext 的一个引用，这样根本不再需要传递任何东西。这种方法的优点是，TStrategy 都可以请求到它所需要的数据。缺点是，TContext 必须对它的数据定义一个更为精细的接口，而且 TStrategy 和 TContext 更紧密地耦合在一起。

2. 让 TStrategy 对象成为可选择的

策略模式在每一时刻只能使用一个策略对象。但如果应用程序启动时，所有的策略对象就已经被创建，那么应用程序就可以在多个策略对象之间进行动态选择。问题是在复杂的系统中，有时 TContext 并不知道策略对象是否已被创建。于是，TContext 在访问某 TStrategy 前先检查它是否存在，如果有，那么就使用它；如果没有，那么 TContext 执行缺省的行为。这种方法的好处是客户根本不需要处理 TStrategy 对象，除非它们不喜欢缺省的行为。

24.3 范例与实践

24.3.1 策略模式在酒店管理系统中的应用

在酒店管理系统中，通常客房的价格不是一成不变的。对于住宿的淡季和旺季、老客户和新客户、散客和团队，都应该有不同的销售策略。显然，销售策略决定了报价。但是基于销售策略的报价体系又不能绑定于某一具体的客户端，因为只有把基于销售策略的报价体系独立出来，才能保证其重用性和可维护性。比如：一种报价体系一方面满足了优惠房价查询、客房结算等多个客户端的使用，另一方面又满足了不断调整的新销售策略的需求，这才算真正做到了重用性和可维护性。

对于以上的设计要求，选用策略模式是最好不过了。策略模式能够让算法变化独立于使

用它的客户端。范例程序是一个基于策略模式的优惠房价查询模块，它包括一个基于销售策略的报价体系和一个优惠房价查询界面。当然，优惠房价查询界面只是该报价体系的客户端之一，报价体系亦可被其他客户端使用。

优惠房价查询模块的设计如图 24-6 所示。它包括了：

- 销售策略类 TSaleStrategy，它是具体销售策略类的抽象基类。
- 3 个具体销售策略类：TVIPStrategy （VIP 卡销售策略）、TTeamStrategy （团队销售策略）、TSeasonStrategy（季节销售策略）。
- 报价类 TPriceContext，它是该策略模式中的上下文，持有一个到 TStrategy 的引用。
- 客户端类 TClient，它是一个窗体类，即房价查询的界面。

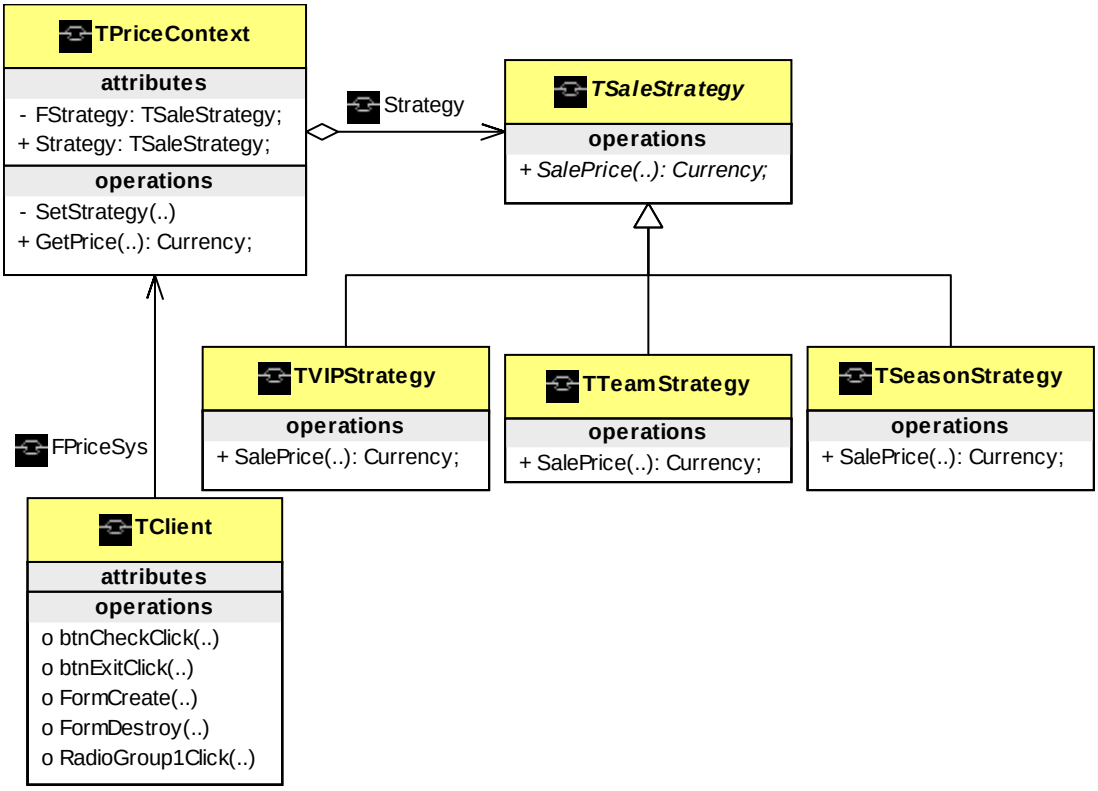


图 24-6 基于策略模式的优惠房价查询模块

示例程序 24-3 是 HotelSaleStrategy 单元的源代码，该单元包含了基于销售策略的报价体系的业务逻辑，用策略模式实现。TSaleStrategy 作为销售策略的抽象基类，其目的是提供一个通用的接口。虚抽象函数 SalePrice 就是这样一个接口。由于 3 个具体销售策略分别是根据季节、VIP 卡、团队人数来制定销售策略的，所以基类接口 SalePrice 的参数设计必须满足 3 个派生类的不同需求。TSaleStrategy 的 SalePrice 函数声明如下：

```
function SalePrice(price:Currency;value:integer):Currency;
                                                    virtual; abstract;
```

它的第一个参数表示传入的固定房价，第二个参数表示传入的优惠条件，该条件因不同的派生类而异。在季节销售策略 TSeasonStrategy 中，该参数表示为入住月份；在 VIP 卡销售策略 TVIPStrategy 中，该参数表示为 VIP 卡的种类；在团队销售策略 TTeamStrategy 中，该参数表示为团队人数。我们发现，这些参数都可以用整数类型，所以在基类中，巧妙地用

一个 value 参数解决了派生类的不同参数需求。这样一来，可以直接让 TPriceContext 将数据放在参数中传递给不同的销售策略类操作，避免了参数冗余。

```
{TPriceContext }
function TPriceContext.GetPrice(price:Currency;value:integer):Currency;
begin
  result:=Strategy.SalePrice(price,value);
end;
```

TPriceContext 在该策略模式中起着上下文作用，它负责引用销售策略对象的不同实例，调用 SalePrice 接口，动态配置具体的折扣算法，并返回实际销售价格。由于有了 TPriceContext 的中介，客户端无需知道具体销售策略是如何实现的；同样，当销售策略进行更新调整时，对客户端程序亦无影响。

示例程序 24-3 HotelSaleStrategy 单元的源代码

```
unit HotelSaleStrategy;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TSaleStrategy = class (TObject)
  public
    function SalePrice(price:Currency;value:integer):Currency;
                    virtual; abstract;
  end;

  TSeasonStrategy = class (TSaleStrategy)
  public
    function SalePrice(price:Currency;value:integer):Currency; override;
  end;

  TVIPStrategy = class (TSaleStrategy)
  public
    function SalePrice(price:Currency;value:integer):Currency; override;
  end;

  TTeamStrategy = class (TSaleStrategy)
  public
    function SalePrice(price:Currency;value:integer):Currency; override;
  end;
```

```
  TPriceContext = class (TObject)
  private
    FStrategy: TSaleStrategy;
    procedure SetStrategy(Value: TSaleStrategy);
  public
    function GetPrice(price:Currency;value:integer):Currency;
    property Strategy: TSaleStrategy read FStrategy write SetStrategy;
  end;

implementation

{TSeasonStrategy }
function
TSeasonStrategy.SalePrice(price:Currency;value:integer):Currency;
begin
  //季节销售策略
  {
  2、3、11 月 8.5 折优惠，
  4、6 月 9 折优惠。
  8、9 月 9.5 折优惠。
  }
  case value of
    2,3,11:result:=price*0.85;
    4,6:result:=price*0.9;
    8,9:result:=price*0.95;
    else
      result:=price;
  end;
end;

{TVIPStrategy }
function TVIPStrategy.SalePrice(price:Currency;value:integer):Currency;
begin
  //VIP 卡销售策略
  {
  0:VIP 银卡 9 折优惠
  1:VIP 金卡 8 折优惠
  2:VIP 钻石卡 7 折优惠
  }

  case value of
    0:result:=price*0.9;
    1:result:=price*0.8;
```

```
    2:result:=price*0.7;
  end;
end;

{TTeamStrategy }
function TTeamStrategy.SalePrice(price:Currency;value:integer):Currency;
begin
  //团队销售策略
  {
  3-5 人团队 9 折优惠；
  6-10 人团队 8 折优惠；
  11-20 人团队 7 折优惠；
  20 人以上团队 6 折优惠。
  }
  result:=price;
  if (value<6) and (value>2) then  result:=price*0.9;
  if (value<11) and (value>5) then  result:=price*0.8;
  if (value<21) and (value>10) then  result:=price*0.7;
  if (value>20) then  result:=price*0.6;
end;

{TPriceContext }
function TPriceContext.GetPrice(price:Currency;value:integer):Currency;
begin
  result:=Strategy.SalePrice(price,value);
end;

procedure TPriceContext.SetStrategy(Value: TSaleStrategy);
begin
  FStrategy:=Value;
end;

end.
```

优惠房价查询模块的客户端程序如示例程序 24-4 所示。该程序提供一个用户选择界面，使得查询者可以任选一种优惠方案。一旦选定优惠条件和公开房价，点击“查询优惠房价”按钮，便得到打折后的优惠价。实际运行效果如图 24-7 所示。

示例程序 24-4 ClientForm 单元的源代码

```
unit ClientForm;

interface

uses
```

```
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls,HotelSaleStrategy, ComCtrls,DateUtils;

type
  TClient = class(TForm)
    RadioGroup1: TRadioGroup;
    btnCheck: TButton;
    btnExit: TButton;
    dtpDate: TDateTimePicker;
    cmbVIP: TComboBox;
    Label1: TLabel;
    Label2: TLabel;
    cmbPrice: TComboBox;
    edtPrice: TEdit;
    Label3: TLabel;
    edtCount: TEdit;
    Label4: TLabel;
    Label5: TLabel;
    Bevel1: TBevel;
    procedure FormCreate(Sender: TObject);
    procedure btnCheckClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
    procedure RadioGroup1Click(Sender: TObject);
  private
    FSeasonStrategy:TSaleStrategy;
    FVIPStrategy:TSaleStrategy;
    FTeamStrategy:TSaleStrategy;
    FPriceSys:TPriceContext;
  public
    { Public declarations }
  end;

var
  Client: TClient;

implementation


{$R *.dfm}

procedure TClient.FormCreate(Sender: TObject);
begin
  FSeasonStrategy:=TSeasonStrategy.Create;
```

```
  FVIPStrategy:=TVIPStrategy.Create;
  FTeamStrategy:=TTeamStrategy.Create;
  FPriceSys:=TPriceContext.Create;
  with cmbVIP.Items do
  begin
    Add('VIP 银卡');
    Add('VIP 金卡');
    Add('VIP 钻石卡');
  end;
  with cmbPrice.Items do
  begin
    Add('甲类标准间 300 元');
    Add('乙类标准间 500 元');
    Add('贵宾间 800 元');
    Add('商务套房 1000 元');
    Add('豪华套房 2000 元');
  end;
end;

procedure TClient.btnCheckClick(Sender: TObject);
var
  i:integer;
  price:Currency;
begin
  case RadioGroup1.ItemIndex of
    0:begin
        FPriceSys.Strategy:=FSeasonStrategy ;
        i:=MonthOf(dtpDate.DateTime);
      end;
    1:begin
        FPriceSys.Strategy:=FVIPStrategy ;
        i:=cmbVIP.ItemIndex;
      end;
    2:begin
        FPriceSys.Strategy:=FTeamStrategy  ;
        i:=StrToInt(edtCount.Text);
      end;
  end;
  case cmbPrice.ItemIndex of
    0:price:=300 ; //甲类标准间 300 元
    1:price:=500 ; //乙类标准间 500 元
    2:price:=800 ; //贵宾间 800 元
    3:price:=1000; //商务套房 1000 元
    4:price:=2000; // 豪华套房 2000 元
```

```
  end;
  edtPrice.Text:=CurrToStr(FPriceSys.GetPrice(price,i));
end;

procedure TClient.FormDestroy(Sender: TObject);
begin
  FPriceSys.Free;
  FSeasonStrategy.Free;
  FVIPStrategy.Free;
  FTeamStrategy.Free;
end;

procedure TClient.btnExitClick(Sender: TObject);
begin
  close;
end;

procedure TClient.RadioGroup1Click(Sender: TObject);
begin
  dtpDate.Enabled:=false;
  edtCount.Enabled:=false;
  cmbVIP.Enabled:=false;
  case RadioGroup1.ItemIndex of
    0:dtpDate.Enabled:=true;
    1:cmbVIP.Enabled:=true;
    2:edtCount.Enabled:=true;
  end;
end;

end.
```

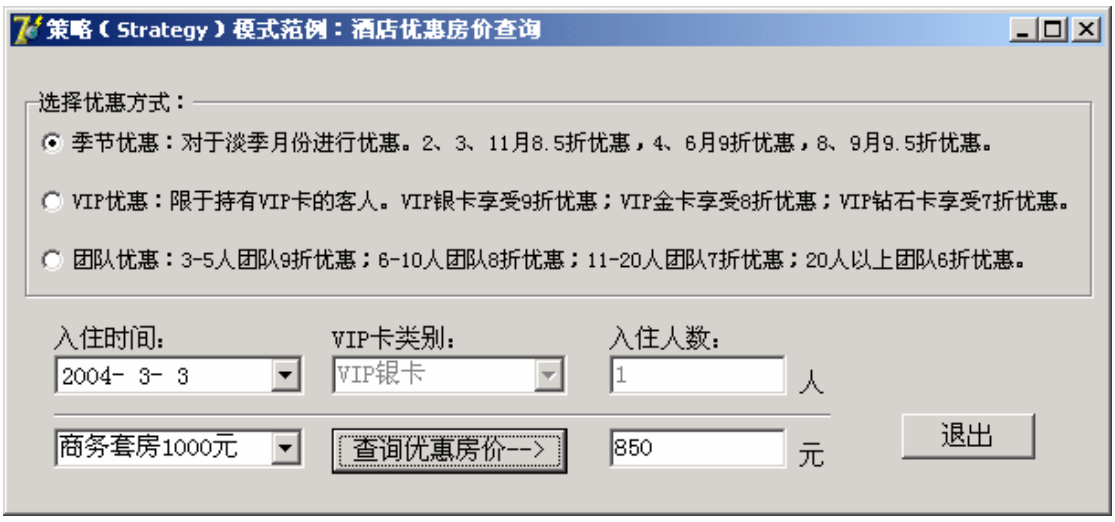


图 24-7 优惠房价查询模块的实际运行界面

24.3.2 范例小结

通过前面范例的演示和剖析，我们进一步讨论策略模式如下：

- **策略模式提供了管理算法集的办法** 策略类的层次结构为 TContext 定义了一系列的可供重用的算法或行为。TStrategy 基类析取出这些算法中的公共功能，派生类通过继承丰富了算法的差异和种类，又避免了重复的代码。
- **策略模式将算法和算法使用环境分离** 如果不将算法和使用算法的上下文分开，直接生成一个包含算法的 TContext 类的派生类，给它以不同的行为，这将会把行为写死到 TContext 中，而将算法的实现与 TContext 的实现混合起来，从而使 TContext 难以理解、难以维护和难以扩展。最后得到一大堆相关的类，它们之间的唯一差别是它们所使用的算法。显然，类的继承关系是强关联，继承关系无法动态地改变算法；而对象的合成关系是弱关联，通过组合策略类对象，使得算法可以独立于使用算法的环境（TContext）而独立演化。
- **使用策略模式可以对大量使用条件分支语句的程序代码进行重构** 当不同的行为堆砌在一个类中时，很难避免使用条件语句来选择合适的行为。将行为封装在一个个独立的策略类中消除了这些条件语句。
- **过多的算法可能会导致策略对象的数目很大** 为了减少系统开销，通常可以把依赖于算法环境的状态保存在客户端，而将 TStrategy 实现为可供各客户端共享的无状态的对象。任何外部的状态都由 TContext 维护。TContext 在每一次对 TStrategy 对象的请求中都将这个状态传递过去。比如范例程序中，我将 TSeasonStrategy 的外部状态入住月份、TVIPStrategy 的外部状态 VIP 卡的种类、TTeamStrategy 的外部状态团队人数都保存在客户端，并通过 TPriceContext 将这些状态传递给销售策略类。这样做的好处是销售策略类变成无状态的了，它们同时可以被客房结算模块等其他模块共享。
- **TStrategy 与 TContext 间的通信开销** 无论各个具体策略实现的算法是简单还是复杂，它们都共享 TStrategy 定义的接口。因此很可能某些具体策略不会都用到所有通过这个接口传递给它们的信息。如果我在范例程序中把 TSaleStrategy 的接口设计成这样：

```
SalePrice(price:Currency;Month:integer;VIP:integer;
                                        Count:integer):Currency;
```

其中的一些参数永远不会被某些具体销售策略类用到。这就意味着有时 TContext 会创建和初始化一些永远不会用到的参数。如果存在这样问题，又无法使用范例程序中的技巧，那么只能在 TStrategy 和 TContext 之间采取紧耦合的方法。